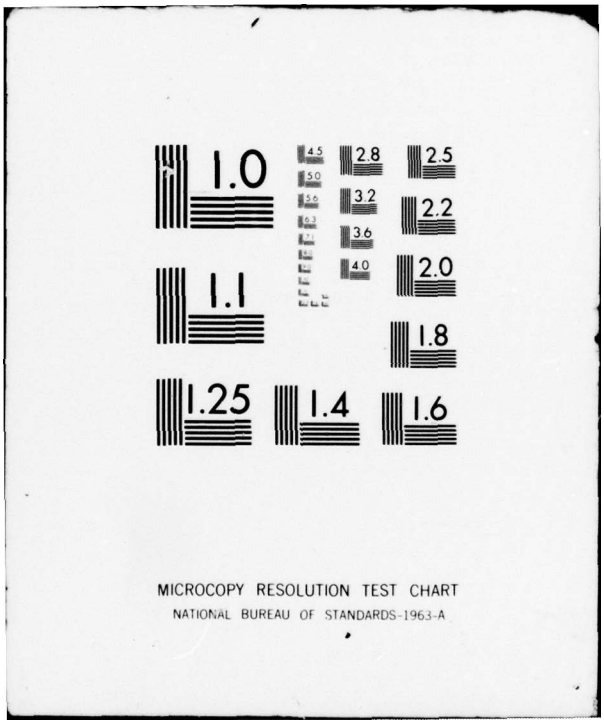
AD-A052 917 AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OHIO SCH--ETC F/G 9/2
A CATEGORIZATION AND EVALUATION OF FORMAL AND SEMI-FORMAL DEFIN--ETC(U)
MAR 78 B D GUILMAIN

UNCLASSIFIED AFIT/GE/MA/78M-1 NL

1 OF 2
AD A052917


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

UNITED STATES AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY
Wright-Patterson Air Force Base, Ohio

(1)

(6) A CATEGORIZATION AND EVALUATION OF FORMAL AND SEMI-FORMAL DEFINITION TECHNIQUES.

(9) Master's THESIS,

(14) AFIT/GE/MA/78M-1 Bruce ~~B.~~/Guilmain
~~Capt USAF~~

(10) Daniel
P105

(11) Mar 78

(12) 117 p.

012 225

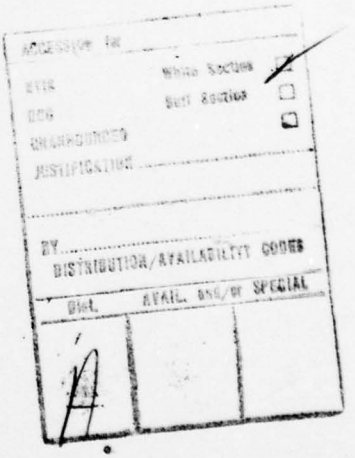
mt

AFIT/GE/MA/78M-1

A CATEGORIZATION AND EVALUATION OF FORMAL AND SEMI-FORMAL DEFINITION TECHNIQUES

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
in Partial Fulfillment of the
Requirements for the Degree of
Master of Science



by
Bruce D. Guilmain, B.S.
Capt USAF
Graduate Electrical Engineering
March 1978

Preface

This thesis establishes a cataloguing criteria, and catalogues three operational formal definition techniques, SEMANOL, the Vienna Definition Language, and BASIS/1-12. The three techniques are then evaluated to determine which technique is the best.

The reasons for cataloguing and finding a fairly simple formal definition technique that completely describes the syntax and semantics of a programming language are many. I think the two most important reasons are standardization and providing information to users. With a formal standard definition of a programming language, hopefully someday a program written in one language can be transfered without worry from one computer system to another. Software systems are becoming more complex and much more expensive each year, and the ability to implement these programs on different computer systems without major revisions is becoming more and more desirable. Language standardization would do much in relieving this problem. A formal definition would not only make transfers easier but also provide useful information about the defined language.

The usefulness of the information provided by a formal definition varies from user to user. Language designers are able to completely understand the implications of their decisions in designing or changing a language. Programmers are able to answer detailed questions about a language without having to guess at the answer and then run several tests verifying the

guessed answer.

I would like to thank Captain Thomas E. Reeves, thesis advisor, for the overall guidance he provided throughout this endeavor. Captain John M. Ives a project officer at Rome Air Development Center provided information on SEMANOL and references of previous evaluations.

I especially want to thank my wife, Carrol, for her support, and understanding which enabled me to complete this course of study.

Bruce D. Guilmain

Contents

Contents

List of Figures

List of Tables

Abstract

Operational techniques for defining computer programming languages are examined; specifically, SEMANOL, the Vienna Definition Language(VDL), and BASIS/1-12. A survey of the operational methods is given, in which specific examples of SEMANOL, the VDL, and BASIS/1-12 are explained in detail. A cataloguing criteria is established and evaluated. The cataloguing criteria is then used to catagorize and evaluate SEMANOL, the VDL, and BASIS/1-12. The SEMANOL technique was judged as the best technique followed by BASIS/1-12 and the VDL, in that order.

A CATEGORIZATION AND EVALUATION OF FORMAL AND SEMI-FORMAL DEFINITION TECHNIQUES

I Introduction

Background

In recent years extensive effort has been spent in developing formal definitions for computer programming languages. A formal definition of a programming language completely describes the syntax(grammar) rules and semantics (meaning) for that language. A formal definition enables language designers to better analyze their work, allows for the standardization of computer programming languages, and provides programmers with a clear and precise understanding of a language. The importance of a formal definition was succinctly stated by J. A. Robinson, "Reliable, high quality computer programming is impossible without a clear and precise understanding of the language in which the programs are written -- this being true quite independently of the merits of the language as a language"(Ref 1). The development of formal definitions of programming languages began with John W. Backus.

John W. Backus developed the Backus Normal Form(BNF), which enables one to give a formal description of syntax rules that describe acceptable sentences in a specific language. A result of BNF is that the syntax of programming

languages has been clearly defined. However, the semantics of a language have not been defined so accurately. Recently other techniques for specifying syntax rules and semantics have been developed. These techniques can be catagorized as either denotational(mathematical) or operational(algorithmetic).

The denotational and operational techniques differ in their basic methodology. The denotational technique uses functional calculus while the operational technique uses the concept of a machine to define the "meaning" of a program. A brief description of the denotational and operational techniques follows.

Denotational techniques do not introduce the concept of a machine. The "meaning" of the source program is described with a functional calculus; that is, the "meaning" of a program is described with an input-to-output function. Since many individuals feel that the true semantics of a source program can not be expressed with an input-to-output function this thesis will focus on the operational techniques.

The operational techniques involve the description of a machine. The specification of a language describes how the machine "behaves" when given a program text and the data required by the program text. The machine will react to an input by either changing states[1] or responding with a compu-

---

[1]A machine state consists of the data set, a source program, and the definition of each instruction. The transfer

tational result. The operational catagory provides two basic methods of defining the "meaning" of a program; compiler oriented methods and interpreter oriented methods

The compiler oriented methods define how source programs may be translated into target programs of a target language with known semantics(Ref 5). The compiler method provides the user with a formal definition of a target language and provides some insight into the relationship between the source and target languages. A drawback of the compiler method is that it does not give the user the definition needed of the source language(Ref 5). The other method in the operational catagory, the interpreter oriented method, defines the "meaning" of a program using an interpreter.

The interpreter oriented method defines, for each source program and its data, an algorithm for computing the value which results from executing the program for the given data(Ref 5). Using the interpreter oriented language definition, and given a program text with the data required by the program text, the interpreter will, depending on the specific method implemented, either change states as a result of the current input statement from the program text, or it will describe the "meaning" of the current statement with a meta-program[2].

---

from one state to another is governed by a "state transition function". The state transition function is defined as the execution of one step(cycle), where several steps are required for the execution of one instruction. The concept of states, in this circumstance, requires the concept of the initial and final(terminating) states. A valid program will begin in the initial state and end in the final state(Ref 12).

[2] A meta-program is a program that defines the steps required to describe the effects of executing either a statement or a program, written in a different language.

Objective

This thesis will focus on the operational methods of defining a computer programming language, and due to the drawback of the compiler methods, mentioned above, only interpreter oriented methods will be examined. The objective of this thesis is to develop a criteria for cataloguing interpreter oriented operational methods for defining computer programming languages; evaluate the established criteria; and finally, catalog interpreter oriented formal definition techniques according to the cataloguing criteria established.

Approach

In establishing a cataloguing criteria, several denotational and operational methods of defining computer programming languages were examined, and other comparative evaluations were studied. Information applicable to interpreter oriented methods was extrapolated from these reports. Examination of several methods of formal definitions gave an insight as to how different methods could vary. A cataloguing criteria was then composed of specific questions aimed at pointing out the variance of different interpreter oriented techniques. The usefulness of the cataloguing criteria was then evaluated. This was accomplished by determining how useful the information gathered from different techniques was to different users. Two groups of users were specified, language designers, and programmers in general. With a cataloguing criteria established, interpreter oriented formal

definition techniques were examined, specifically, TRW's SEMANOL, and IBM's Vienna Definition Language(VDL), and IBM's BASIS/1-12.

A by-product of the above process makes this thesis the only single source for a reader interested in understanding the mechanics of TRW's SEMANOL, IBM's VDL, and IBM's BASIS/1-12.

Overview

Chapter II of this thesis presents the process involved in developing a cataloguing criteria. Chapter III provides an evaluation criteria, and evaluates the cataloguing criteria developed in chapter II. A brief summary of SEMANOL,VDL, and BASIS/1-12 is presented in chapters IV, V, and VI, respectively. In chapter VII the cataloguing criteria developed in chapter II is applied to each technique, SEMANOL, VDL, and BASIS/1-12. The conclusion, chapter VIII, contains the results of the cataloguing procedure along with the advantages and disadvantages of each technique

II A Cataloguing Criteria

Approach

The purpose of a cataloguing criteria is to indicate the quality of a specific technique used to formulate a formal definition of a programming language, and enumerate differences among various techniques. Several methods were studied to determine what qualities should be examined when evaluating formal definition techniques. The research involved the denotational and operational techniques listed below.

The denotational techniques examined were W-grammars(Ref 3), production systems with an axiomatic approach to semantics (Ref 3), attribute grammars(Ref 3), an axiomatic approach by Hoare(Ref 10), lambda-calculus as used by Landin(Ref 11), and the Oxford Definition Method(Ref 1;4).

The operational techniques studied were SEMANOL(Ref 2;4;7;8;9), the Vienna Definition Language(VDL)(Ref 3;5;6;12), and BASIS/1-12(Ref 16).

Examination of the denotational and operational techniques provided the author with information relating what each technique considered primary characteristics of a programming language.

Previous comparative evaluations of denotational and operational techniques were then studied, specifically, "The Definition of Programming Languages"(Ref 1), and "A Sampler of Formal Definitions"(Ref 3). Both of the comparative evaluations examined operational and denotational techniques against each

other. Such evaluations are useful in pointing out the differences between the two basic approaches. A reader searching for any formal definition technique, regardless of approach, would find such evaluations useful.

However, if one is interested in interpreter-oriented techniques, then an evaluation of only interpreter-oriented techniques is required. Evaluating two distinct catagories is inequitable. "Weakness" inherent in a catagory should not be listed as a weakness of a specific technique; a particular interpreter used in an interpreter-oriented technique should not be classified as a weakness. An evaluation of a specific formal definition technique should not evaluate techniques from the denotational catagory and the operational catagory together.

The above research indicated major areas of concern when defining a computer programming language. These major areas are:

A. Control. Specifically, how does this formal definition technique describe the flow of control for sequences, jumps, loops, and procedure calls.

B. Memory. This area involves the storage of values for future reference.

C. Expressions. Specifically, the rules governing the evaluation of expressions.

D. Clarity. Clarity in this field of study refers to how well the user of a formal definition technique is able to understand the technique.

E. Completeness. Completeness refers to any weakness in

the definition technique. The formal definition of a language should be able to describe every syntax or semantic event which may occur in that language.

Any cataloguing criteria which proposes to give some indication of the quality of a formal definition technique must at a minimum address these areas.

Realizing that in any evaluation a certain amount of subjectivity is introduced, a criteria composed of specific questions was formulated. Specific questions minimize subjectivity, and make subjective answers more apparent to the reader. This technique was particularly useful in dealing with the clarity of a definition. A specific question aimed directly at clarity would rely only on a user's unsupported opinion for an answer. However, several questions aimed at characteristics which make a specific method easier to understand enables one to reduce the amount of subjectivity involved in deciding the clarity of a technique.

The following question is an example of the type of questions used in determining the clarity of a definition. Does this method seperate context-free syntax, context-sensitive syntax, and the semantic parts of a language definition? Though this question does not seem to be directed toward clarity, the seperation of context-free syntax, context-sensitive syntax, and the semantics of a definition makes the method easier to understand and thus the technique is clearer than if these parts were not seperated. Several other questions relating to the clarity of a technique are listed with the other questions used

in the cataloguing criteria, at the end of this chapter.

The concept of completeness is almost impossible to decompose into components as was done with clarity and so the subjectivity involved in determining this characteristic could not be deminished.

Characteristics which do not particularly fit into the above areas, but are vital in distinguishing one formal definition from others, include:

A. Context-free and context-sensitive syntax. How does a particular method process or evaluate the syntax?

B. Defining the execution of a program. Does the method use the machine-theoretic approach in which execution is described by changing states until the final state is achieved, or is the execution described with a meta-program(Ref 7).

C. Deciding the validity of a program. How demanding is the definition? When will a method declare a program valid (after a good syntax check, a good semantics check, or proper execution of the program)(Ref 3)?

D. Specification of implementation dependencies. This area pertains to a definition which is implemented on a computer and deals with finding out how the constraints of the host computer are defined within the definition.

Criteria

A cataloguing criteria which covers each of the above characteristics enables one to obtain an idea of the quality of a specific formal definition technique, and enables a reader to

compare various formal definition techniques. Each of the qualities specified above are listed below, and then followed by the specific questions used in the cataloguing criteria to determine if a specific quality was present in the technique being evaluated.

No "ideal" answers are provided. The evaluation of different techniques involves evaluating the techniques directly against each other, not against an "ideal" technique, and then against each other.

I Control

A. How does this method process the flow of control for sequences, jumps, loops, and procedure calls?

II Memory

B. How is memory defined (assignment statements, variables, etc.)?

III Expressions

C. How are evaluations of expressions defined?

D. How are lexical transformations processed?

E. Can external functions and relations be called from a system library?

F. How is the semantic operator expressed?

IV Clarity

G. Can people understand the method?

H. Is high-level expressiveness utilized? (How much detail must one know before utilizing the technique?)

I. Are the mneumonic names helpful to the reader?

J. Does this method seperate context-free syntax, context-sensitive syntax, and the semantic parts of a language definition?

V Completeness

K. Does this technique provide a complete definition?

VI Context-free and context-sensitive syntax

L. How is the context-free syntax processed?

M. How is the context-sensitive syntax processed?

VII Defining the execution of a program

N. How does this method define the execution of a program?

VIII Deciding the validity of a program

O. What constitutes a valid program?

IX Specification of implementation dependencies

P. How are the implementation dependencies defined?

Q. Are the representations of the data types and operators machine independent?

The above cataloguing criteria specifies what was examined in cataloguing different interpreter-oriented formal definition techniques. The following chapter conducts an evaluation upon the above criteria to determine its usefulness.

III Evaluation of the Established Cataloguing Criteria

A cataloguing criteria must cover every major requirement of a formal definition technique and display qualities of the technique that are beneficial to a wide range of users. This chapter evaluates the established cataloguing criteria given in chapter II to determine if it conforms to the above description.

Constructing the cataloguing criteria in chapter II consisted of identifying major areas of importance(control, memory, expressions, clarity, and completeness) in a formal definition. Questions aimed at exploiting how different formal definition techniques cover these areas were then formulated. Thus the first goal of this evaluation, verifying that the cataloguing criteria covers the important areas in a language definition, has been confirmed, since these areas were used in the construction of the criteria.

The second goal of the cataloguing criteria is to display qualities of a formal definition technique that are beneficial to a wide range of users. For this evaluation, two groups of users were identified in order to represent both ends of a spectrum of users. The two groups of users were language designers and programmers in general.

The value of the established cataloguing criteria to language designers was determined by listing several qualities of importance to language designers interested in changing a

language. Each listed quality was then followed by related questions from the cataloguing criteria. Qualities of importance to language designers contemplating a change in a language are:

A. Changes should be easy to implement. The formal definition should be easy to use.

B. The effects of a particular source program statement should be easy to trace.

C. Effects of a language change should be completely defined.

D. The effects of a language change on the flow of control should be easy to trace.

E. Context-free syntax and context-sensitive syntax should be easy to modify.

F. The user should be able to verify the interface between the language and external system functions.

Evaluation of how well the established criteria verified these qualities in a formal definition was determined by listing each of the qualities followed by related questions from the cataloguing criteria. These qualities and the related questions are listed below.

<u>Changes should be easy to implement. The formal definition should be easy to use.</u>

a. Can people understand the method?

b. Is high-level expressiveness utilized? (How much detail must one know before utilizing the technique?)

c. Are the mneumonic names helpful to the reader?

d. Does this method seperate context-free syntax, context-

sensitive syntax, and the semantic parts of a language definition?

The effects of a particular source program statement should be easy to trace.

a. How is the semantic operator expressed?

b. Can people understand the method?

c. Is high-level expressiveness utilized? (How much detail must one know before utilizing the technique?)

d. Are the mneumonic names helpful to the reader?

e. Does this method seperate context-free syntax, context-sensitive syntax, and the semantic parts of a language definition?

f. Does this technique provide a complete definition?

Effects of a language change should be completely defined

a. Can external functions and relations be called from the system library?

b. Does this technique provide a complete definition?

c. What constitutes a valid program?

The effects of a language change on the flow of contrc should be easy to trace.

a. How does this method process the flow of control for sequences, jumps, loops, and procedure calls?

b. How are evaluations of expressions defined?

c. How is the semantic operator expressed?

d. Does this method seperate context-free syntax, context-sensitive syntax, and the semantic parts of a language definition?

Context-free syntax and context-sensitive syntax should be easy to modify.

a. How is the semantic operator expressed?

b. Does this method seperate context-free syntax, context-sensitive syntax and the semantic parts of a language definition?

c. How is the context-free syntax processed?

d. How is the context-sensitive syntax processed?

The user should be able to verify the interface between the language and external system functions.

a. Can external functions and relations be called from the system library?

As shown above, the established cataloguing criteria provided information related to every area of interest to a language designer.

At the other end of the user spectrum is the programmer. The programmer is interested in answering specific questions pertaining to "how" a language "works". Specific areas of concern, to programmers, are:

A. The definition should be easy to understand;

B. One should easily see how and what a source program statement does;

C. The flow of control within a source program should be easy to follow;

D. The effects of source program statements on each other should be apparent;

E. The requirements of the syntax and semantics should be

easy to determine;

F. The user should easily understand how values are stored;

G. The user should not have to worry with items that are not relevant;

H. One should be able to determine exactly when a substitution takes place.

The programmer should select a formal definition technique that fulfills these eight areas of concern. The usefulness of a cataloguing criteria, to a programmer, could be determined by examining how many of the above eight areas are evaluated. Evaluation of the cataloguing criteria, verifying inspection of the eight areas, is listed below. The evaluation consists of the quality desired by the programmer followed by the related questions.

<u>The definition should be easy to understand.</u>

a. Can people understand the method?

b. Is high-level expressiveness utilized?

c. Are the mneumonic names helpful to the reader?

<u>One should easily see how and what a source program statement does.</u>

a. How are the evaluations of expressions defined?

b. How is the semantic operator expressed?

<u>The flow of control within a source program should be easy to follow.</u>

a. How does this method process the flow of control for sequences, jumps, loops, and procedure calls?

The effects of source program statements on each other should be apparent.

a. How does this method define the execution of a program?

The requirements of the syntax, and semantics should be easy to determine.

a. Does this method seperate context-free syntax, context-sensitive syntax, and the semantic parts of a language definition?

b. How is the context-free syntax processed?

c. How is the context-sensitive syntax processed?

The user should easily understand how values are stored.

a. How is memory defined (assignment statements, variables, etc.)?

The user should not have to worry with items that are not relevant.

a. Can external functions and relations be called from the system library?

b. How are the implementation dependencies defined?

c. Are the representations of the data types and operators machine independent?

One should be able to determine exactly when a substitution takes place.

a. How are lexical transformations processed?

The cataloguing criteria does supply information to the programmer, in every area of interest to the programmer.

This evaluation has shown that the cataloguing criteria

exposes information in the five major areas of concern in a formal definition (control, memory, expressions, clarity, and completeness). The evaluation also has shown that the cataloguing criteria provided useful information in every major area of concern for both language designers and programmers in general.

The result of this evaluation reveals that the established cataloguing criteria provided all the information normally required by a cataloguing criteria. Therefore, it is the opinion of this writer that the established cataloguing criteria is at a minimum, satisfactory.

With a satisfactory cataloguing criteria established the process of examining SEMANOL, the Vienna Definition Language, and BASIS/1-12 can begin. The following chapter, chapter IV, provides a brief introduction to the SEMANOL technique.

IV SEMANOL

Background

Research, that lead to the SEMANtics Oriented Language (SEMANOL), began at TRW by Dr. E. K. Blum in the late 1960's. SEMANOL was developed to enable a user to completely describe the syntax and semantics of procedural programming languages, such as ALGOL 60, FORTRAN, COBOL, and SIMULA 67. Designed to provide a definition used by people, SEMANOL provides a precise method of communicating syntax and/or semantic details of a computer programming language. The SEMANOL specification can be processed by a SEMANOL interpreter, thus, a SEMANOL language definition can be machine tested(Ref 2).

Methodology

A SEMANOL meta-program provides a formal technique for defining the "execution" of a source program. The effect of executing a source program, written in the defined language, is obtained by determining the effects of each computation separately. Obtaining each computation separately is accomplished through a parse tree representation of the source program. The leaves(terminal nodes) of the parse tree consists of the source program text. An example of a parse tree, for the statement "LET A=B+C", with the grammar rules given below, is shown in Fig. 1. The grammar rules are:

<LINE> ::= <STATEMENT>

<STATEMENT> ::= 'LET ' <NUMERIC-VARIABLE> '=' <NUMERIC-EXP>

<NUMERIC-VARIABLE> ::= 'A' | 'B' | 'C'

<NUMERIC-EXP> ::= <NUMERIC-VARIABLE> '+' <NUMERIC-VARIABLE>

Where:

"::=" means the item on the left side is composed of the items on the right side, and

"< >" indicate that the quantity inside is to be defined; that is, it must appear on the left side of "::=" for at least one grammar rule.

"|" as specified in the third grammar rule indicates an "exclusive or" condition. Either the 'A', the 'B', or the 'C' can compose a NUMERIC-VARIABLE.

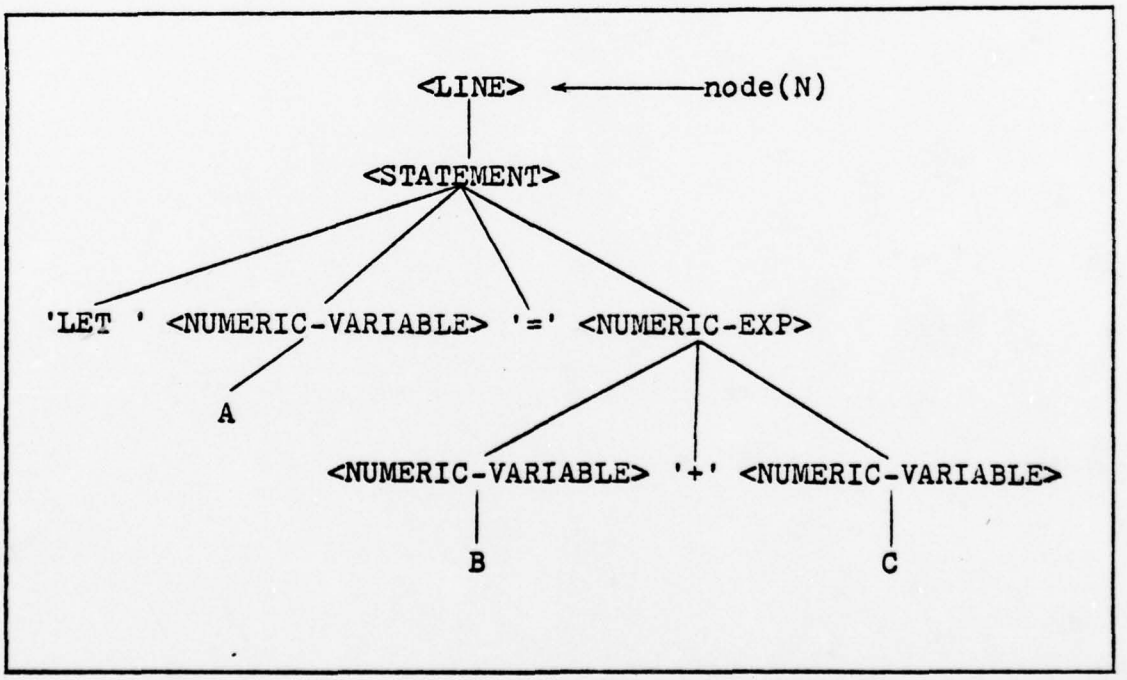


Fig. 1. Parse tree for the statement "LET A=B+C".

The concept behind the SEMANOL meta-program is defined below. The SEMANOL system defines a programming system (S) as $S = (P, I, T, \bar{\Phi})$, where

P = The set of programs which can be expressed in the programming system;

I = The set of input values;

T = The set of output traces. A trace is an ordered record of significant actions(such as assignment) that are performed by the program as it is executed; it is the visible manifestation of performing the algorithm that is the operational SEMANOL specification of semantics; and

$\bar{\Phi}$ = The semantic operator. This operator, given as $\bar{\Phi}: P \times I \rightarrow T$, is considered to define the "meaning" of a program(Ref 2).

The product of two sets P and I (P x I) is defined as all the finite set of pairs pi such that p is in the set P and i is in the set I. Individual members of P, I, and T are represented by p,i, and t, respectively. The execution effects of a given program, p, can be represented as

$$\bar{\Phi}(p,i) = t \qquad (1)$$

which states, the effect of executing the program p with an input i is the output trace t.

A SEMANOL meta-program p_S describes the semantics of the defined language. The effect of the semantic operator $\bar{\Phi}$ equals the effect of the SEMANOL meta-program defining $\bar{\Phi}$, thus,

$$p_S(p,i) = \bar{\Phi}(p,i) = t \qquad (2)$$

The complete SEMANOL system involves an interpreter, capable of executing the SEMANOL meta-program. The semantic effects of the SEMANOL meta-program are defined by a semantic operator for the SEMANOL system ($\bar{\delta}_S$) thus,

$$\bar{\delta}_S(p_S,(p,i)) = \bar{\delta}(p,i) = t \qquad (3)$$

While the SEMANOL approach is algorithmic, it is also "program-oriented". SEMANOL does not use the basic concept of machine states, usually associated with interpreter-oriented techniques. However, SEMANOL can be considered interpreter-oriented because it employs the concept of a machine, and neither the SEMANOL meta-program nor the source program, written in the defined language, are translated into another language(Ref 2;5;7).

The Meta-language

SEMANOL was developed to provide the user with an easily understood method for defining semantics. The use of FORTRAN, or any other language, would have required complex programs to define the semantics. These programs would have been less readable, and less comprehensible.

The SEMANOL description of a language consists of a program with four sections. The four sections are: the declaration section, the context-free section, the semantic section, and the control section.

The first section, the declaration section, identifies SEMANOL declared global variables, and syntactic components. Declared global variables are independent of any individual

source program. Therefore, declared global variables are limited to variables that retain control related information, such as, a list of return points from subroutines, etc..

Syntactic components represent a very interesting characteristic of SEMANOL. Executing a program, written in the defined language, first involves parsing the program. The program text serves as the terminal nodes of the parse tree. Each parse tree non-terminal node has several attributes. One of the attributes, syntactic component, is composed of a pair (s,v), where s is a semantic definition name, and v is a SEMANOL value assigned to that name. The first time a node is used in a semantic definition, a SEMANOL value is obtained. If that value must remain constant throughout the execution of the source program, that value is assigned to the "v" in the syntactic component. At a later time in the program if that specific node is used in the same semantic definition, the value is available and will not have to be recomputed. The semantic definition "Left-Hand-Side-Of" is an example of the type of semantic definition used in a syntactic component. Applying "Left-Hand-Side-Of" to node(N), in Fig. 1, which has a value "LET A=B+C", will always obtain the value "A". The next time "Left-Hand-Side-Of" is applied to node(N) the value "A" will be available and will not have to be derived again.

The total number of declared global variables and syntactic components specified in the declaration section is a function of the language defined.

The second section is the context-free section. This section

contains the syntactic definitions which specify an (almost) context-free grammar over the ASCII character set for the defined language. The only context-sensitive feature implemented in this section is the keyword #GAP. The keyword #GAP represents the set of zero or more ASCII blank characters(one or more when concatenated with the left or right sides of alphanumeric strings). Grammars are used as recognition grammars in SEMANOL and the above feature ensures that a valid program, syntactically, will be recognized.

If a grammar specified in this section is ambiguous, an error message results. An ambiguous grammar is a grammar in which two different parse trees have identical leaves. In this situation an ambiguous grammar is exposed by the existence of two different parse trees for the same source program. proving a grammar unambiguous is essentially impossible, unless that grammar is composed of relatively few rules(few enough to test every combination of rules). Therefore, the flaw of having specified an ambiguous grammar, in the language definition can only be recognized by parsing a program that causes more than one parse tree to be generated.

The semantic definitions section is the third section. "Semantic definitions consist of a "semantic definition name", followed by an optional "dummy parameter list" followed by a semantic definition body. A semantic definition may be functional or procedural, i.e. it specifies the selection of a SEMANOL expression which is to be evaluated(functional), or it specifies a sequence of "statements" which are to be executed in order

(procedural)(Ref 14)."

The fourth section in a SEMANOL description of a programming language is the control section. This section contains a sequence of statements which are executed in order. Execution of a SEMANOL program begins with the first statement in the control section, and proceeds until the effects of executing th program, written in the defined language, are completed.

Execution

The previous section described the SEMANOL program used to obtain the effects of executing a source program. This section describes the processes the SEMANOL program performs upon the source program.

A lexical analysis, dependent on the defined language, is the first process performed upon the source program text. During this process any lexical substitutions(macro substitutions) are performed, and comment statements removed. An example of a macro substitution is the "DEFINE" statement in JOVIAL(J3). The "DEFINE" statement "DEFINE EXP "A+B*C" specifies that everywhere "EXP" appears in the program text "A+B*C" must be substituted. Therefore, when processing a JOVIAL program a lexical analysis must examine the text for "DEFINE" statements.

The source program is then parsed using the grammar specified in the context-free section. After a successful parse, several context-sensitive tests are performed. The number of context-sensitive tests is a function of the language defined with the SEMANOL program. The MINIMAL BASIC specification contains

seventeen context-sensitive tests. Examples of the context-sensitive tests used in the MINIMAL BASIC specification are: testing to verify that all line numbers are non-zero, testing to verify that all line numbers are uniquely numbered, and testing to verify that all "FOR" statements have matching "NEXT" statements, etc..

The SEMANOL program then computes the effect of executing the syntactically valid program, using the semantic definitions section. The above processes are shown in a flow chart format in Fig. 2.

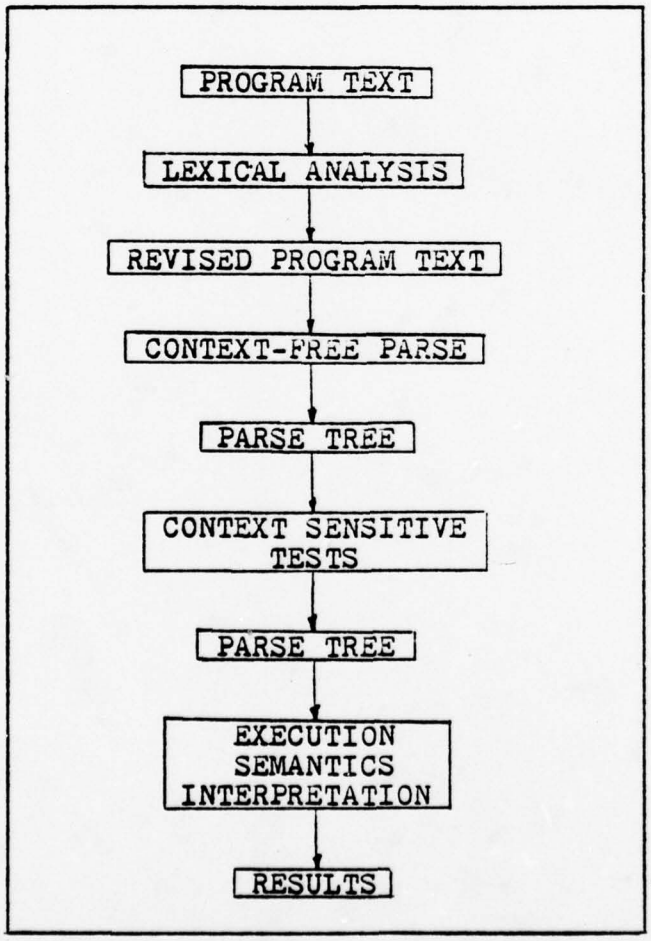


Fig. 2. Stages of a definition.

During the above processes the occurance of any errors either in the source program text or in the SEMANOL program description results in the output of the appropriate error message(Ref 8).

An example of a SEMANOL definition is given below. The language defined is very simple, enabling the reader to understand the four sections in the definition. The defined language the Assignment Statement Language, ASL consists of assignment statements, a stop statement, and an end statement. Each statement except the end statement is terminated with a semicolon. The format of the assignment statement consists of 'LET" followed by a single alphabetic character(A, B, or C), then "=" followed by an unsigned integer constant(LET A = 2).

The words composed of capital letters, in the definition, are SEMANOL commands. These commands are not defined since their meaning is obvious. The other words linked with hyphens are entities defined in the semantic section or in the control section. SEMANOL definitions, in the syntactic and semantic sections, begin with "#DF" and end with "#.", and "#U" indicates an "exclusive or" condition. An example of a SEMANOL definition is given below.

```
#DF sequence-of-statements-in(basic-program)
=> #SEQUENCE-OF <numeric-let-statements>  #U <stop-statement>
   #U <end-statement>  #IN basic-program  #.
```

Treating the SEMANOL definition of the test language as a program is helpful since using the definition involves executing the statements in the control section. The SEMANOL definition

for the specified language is listed below, a few comments, enclosed in quotation marks, are included.

DECLARATION SECTION

Rule

1. #DECLARE-GLOBAL:
   basic-program
   current-statement #.
2. #DECLARE-SYNTACTIC-COMPONENT:
   sequence-of-statements-in
   is-not-stop-or-end
   left-hand-side-of
   right-hand-side-of #.

CONTEXT-FREE SECTION

3. #DF<program> => <%<line>><end-statement> | <%<line>>
   <stop-statement><end-statement>
4. #DF<line> => <numeric-let-statement><#GAP>';'
5. #DF<numeric-let-statement> => 'LET'<#GAP><numeric-variable><#GAP>'='<#GAP><numeric-constant>#.
6. #DF<numeric-variable> => 'A' | 'B' | 'C' #.
7. #DF<numeric-constant> => <%1<digit>> #.
8. #DF<digit> => '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9' #.
9. #DF<end-statement> => 'END' #.
10. #DF<stop-statement> => 'STOP' <#GAP>';'

"Where "%" means the following item is repeated zero or more times, "%1" means the following item is used at least once."

SEMANTIC-SECTION

Rule

1. #DF is-context-free-syntactically-valid(basic-program)
   => #TRUE #IF basic-program #IS-NOT #UNDEFINED
   => #FALSE #OTHERWISE #. "an error message results"

2. #DF sequence-of-statements-in(basic-program)
   => #SEQUENCE-OF <numeric-let-statement> #U <stop-statement> #U <end-statement> #IN basic-program #.

3. #DF is-not-stop-or-end(stmt)
   => #TRUE #IFF #NOT stmt #IS <stop-statement> #U <end-statement>
   => #FALSE #OTHERWISE #.

4. #DF effect-of(current-statement)
   => #BEGIN
      #COMPUTE! #ASSIGN-LATEST-VALUE(left-hand-side-of(current-statement), "receives" right-hand-side-of(current-statement))
      #END #.

5. #DF left-hand-side-of(current-statement)
   => SEG 3 #OF current-statement #.
   "the third item in the syntax definition of the numeric-let-statement, the numeric-variable"

6. #DF right-hand-side-of(current-statement)
   => SEG 7 #OF current-statement #.
   "the seventh item in the syntax definition of the numeric-let-statement, the numeric-constant"

CONTROL COMMANDS SECTION

```
Rule
  1.   #CONTROL-COMMANDS:
  2.   #ASSIGN-VALUE! basic-program = #CONTEXT-FREE-PARSE-
        TREE( #GIVEN-PROGRAM, program)
  3.   #IF($basic-program$) is-context-free-syntactically-valid
  4.   #THEN
  5.   #BEGIN "The following loop determines the execution
              effects of each statement in the source program
              text".
  6.        #ASSIGN-VALUE! current-statement = #FIRST-ELEMENT-
             sequence-of-statements-in(basic-program)
  7.        #WHILE($current-statement$) is-not-stop-or-end #DO
  8.           #COMPUTE! effect-of(current-statement)
  9.   #END
 10.   #COMPUTE! #STOP  #.
```

A test program demonstrating the previous SEMANOL definition, of the language used to write this program, is given below.

```
LET A = 1;
LET B = 2;
END
```

The reader "executing" this test program must start with the first control command, in the SEMANOL definition, which is, "ASSIGN-VALUE! basic-program = #CONTEXT-FREE-PARSE-TREE(# GIVEN-PROGRAM,program). This command parses the source program text, using the context-free syntax section. The basic-program resulting from this command is shown in Fig. 3. The next con-

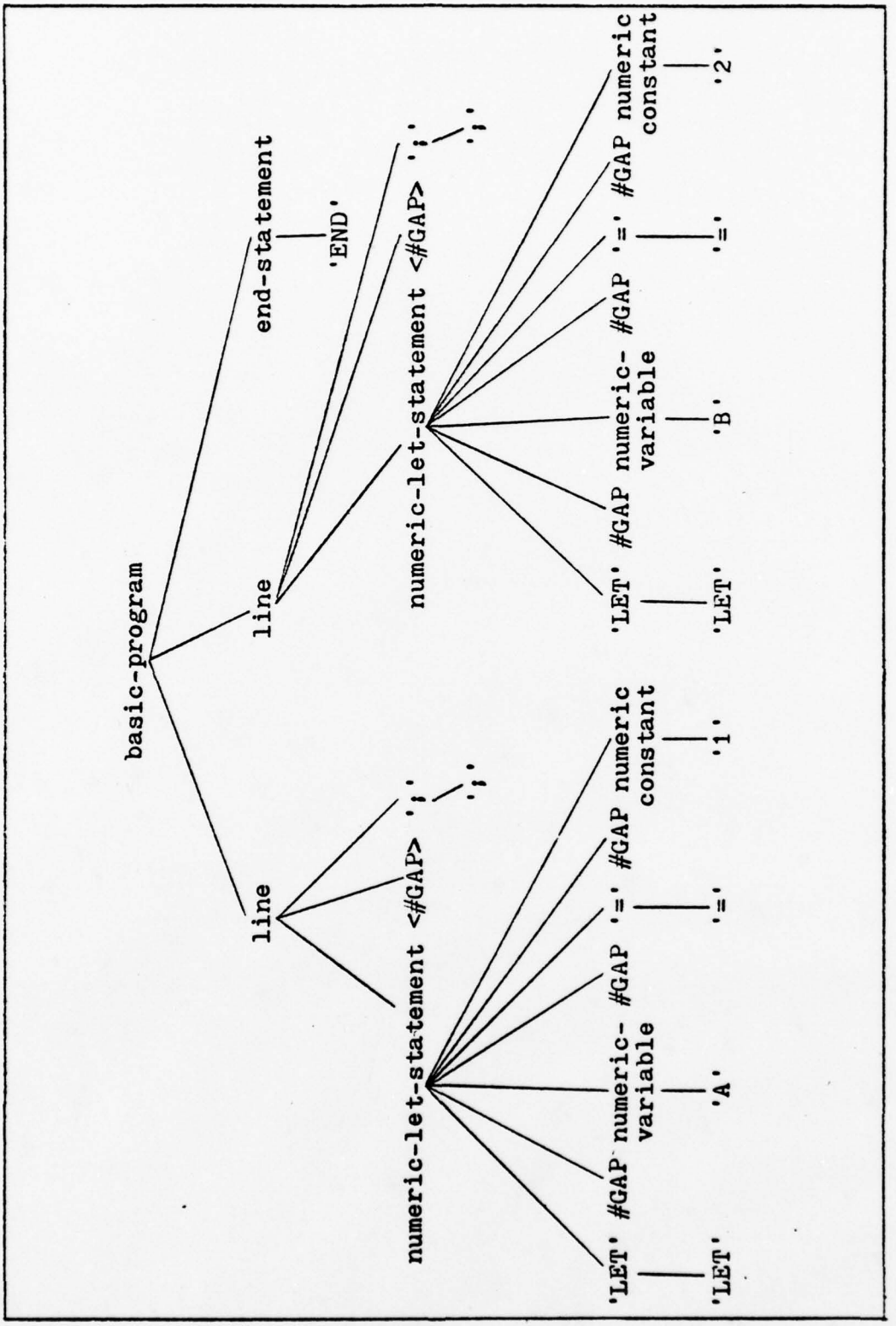


Fig. 3. Context-free parse of the source program.

trol command "#IF($basic-program$) is-context-free-syntactically-valid #THEN", verifies that a valid parse tree was obtained from the previous command. A valid parse tree in the language definition, will allow the execution of the following commands, a loop delimited by "#BEGIN" and "#END", which determines the execution effects of the test program. The first command within the loop is "#ASSIGN-VALUE! current-statement = #FIRST-ELEMENT-IN sequence-of-statements-in(basic-program)". This command takes the first available statement from the source program text(LET A = 1) assigns it to current-statement and then deletes it from the source program text. The test program now looks like

```
LET B = 2;
END
```

The "#WHILE($current-statement$) is-not-stop-or-end #DO" command is execute, verifying that the entire source program has not been executed. If the current statement is not a "stop" or an "end" statement the "#COMPUTE effect-of(current-statement)" command is executed. The "meaning" of "effect-of" is defined in the fourth definition in the semantic section(page 29). Essentially, the value "1" is assigned to A. The next control command executed is the first one in the loop, "ASSIGN-VALUE! current-statement = #FIRST-ELEMENT-IN sequence-of statements-in(basi-program)". This time "LET B = 2" is assigned to the current statement. The loop continues until a "stop" or "end" statement is obtained in the current statement. When the "end"

statement in the test program is reached the effects of executing the test program will have been realized.

This chapter defined the SEMANOL formal definition technique, and provided a simplistic example of a SEMANOL description of a language.

With a brief introduction to SEMANOL complete, the process of examining the Vienna Definition Language(VDL) can begin. The following chapter, chapter V, provides a brief introduction to IBM's Vienna Definition Language.

V The Vienna Definition Language

Background

The Vienna Definition Language(VDL) is a product of research conducted in the late 1960's at IBM's laboratory in Vienna, Austria. The objective of the research was to develop a technique to produce a formal definition of Programming Language I(PL/I). The developed technique was to be capable of defining a programming language(syntax and semantics) without ambiguity, with the ultimate goal of standardizating the defined language.

Methodology

The Vienna Definition Language is an interpreter oriented programming language capable of defining programming languages. The VDL definition process utilizes three abstract machines, an ANALYZER, a TRANSLATOR, and an INTERPRETER.

A source program is not interpreted directly, but preprocessed by the ANALYZER and the TRANSLATOR. This preprocessing, similar to SEMANOL's lexical analysis, context-free syntax parse, and the context-sensitive syntax tests, translates the source program into an abstract object representation that is ready for interpretation. The goal of this chapter is to present a brief informal introduction to the Vienna Definition Language, covering the data structure the language manipulates, and how the semantics of a program are defined(Ref 3;5;6;12)

The Meta-language

The Vienna Definition language defines the semantics of a program by manipulating the data structure, a tree, that represents the source program. This tree manipulation is accomplished with operators that select tree components, construct new trees, and assign new values to nodes of existing trees(Ref 5).

A source program ready for interpretation is represented by abstract data objects. There are two classes of data objects: elementary objects, having no components and represented by terminal nodes on a tree; and composite objects composed of a finite number of data objects, and represented by nonterminal nodes on a tree. An elementary object "E" is shown in Fig. 4, and a composite object "C1" is shown in Fig. 5.

A very important feature of the "Vienna tree" is that each branch(a connector and a node, coming from a node) is labeled. The labeling of each branch allows one to represent an entire tree with a set of "selector-object" pairs such as <S1:E1>. The selector pair <S1:E1> indicates that S1 is the selector of object E1, as shown in Fig. 5. The object E2 in Fig. 5 would be selected with <S1:(<S2:C1>)>, where <S2:C1> indicates object C2 and then <S1:C2> selects object E2. The Vienna tree depicted in Fig. 5 is represented by expression 1.

$$C1 = (\langle S1:E1\rangle, (\langle S2:(\langle S1:E2\rangle, \langle S2:E3\rangle)\rangle), \langle S3:E4\rangle) \qquad (1)$$

Every branch emanating from the same node must have a unique label, and selecting a branch which does not exist(selecting S4 of node C1) yields the null object.

The Vienna Definition Language operators, described below, manipulate the Vienna tree. There are three basic catagories of operators in the VDL, the construction operator u_o, the mutation or assignment operator u, and relational operators.

The construction operator u_o allows one to "build" a Vienna tree. An example of the construction operator in operation is expression 2.

$$u_o(<S1:E1>, (<S2:(<S1:E2>, <S2:E3>)>), <S3:E4>) \qquad (2)$$

The execution of expression 2 results in the creation of the Vienna tree specified in Fig. 5. Thus the construction operator, u_o, is used to build new objects.

The mutation or assignment operator u is used to modify existing objects. An example of the mutation operator's capability is demonstrated with expression 3.

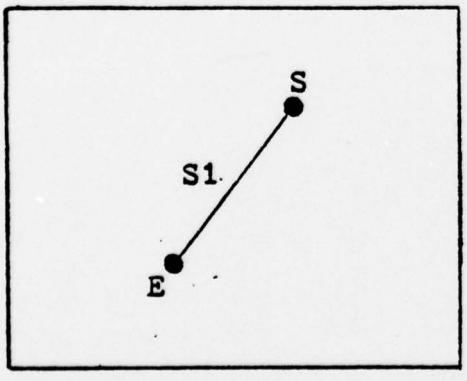


Fig. 4. Elementary Object

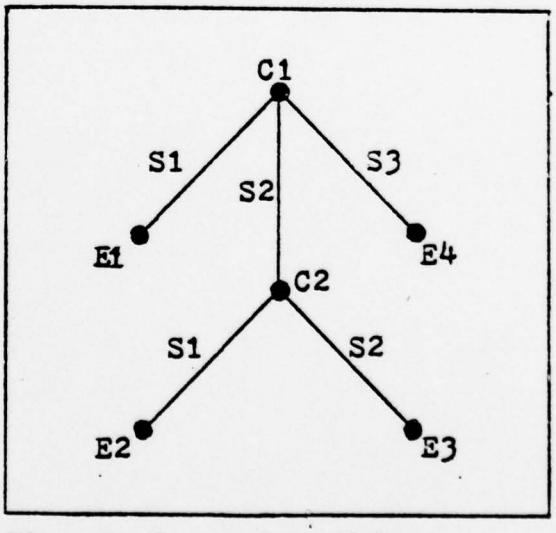


Fig. 5. Composite Object

$$u(C1:\langle S1:E0\rangle) \qquad (3)$$

Expression 3 results in a mutation of object C1; branch S1, if existent, is deleted, and a new branch S1 with an object E0 is added. Applying expression 3 to Fig. 5 results in a modified tree, shown in Fig. 6.

The relational operators, predicates, are used to identify different sets of objects. A predicate can be regarded as the name of an object set. The value of a predicate is either TRUE or FALSE. An example of a predicate is "IS-DIGIT" which identifies the set of digits. The predicate "IS-DIGIT" is defined in expression 4.

$$\text{IS-DIGIT} = 1 \text{ v } 2 \text{ v } 3 \text{ v } 4 \text{ v } 5 \text{ v } 6 \text{ v } 7 \text{ v } 8 \text{ v } 9 \text{ v } 0 \qquad (4)$$

where "v" signifies a logical exclusive or condition. Applying "IS-DIGIT" to any digit would yield the value TRUE.

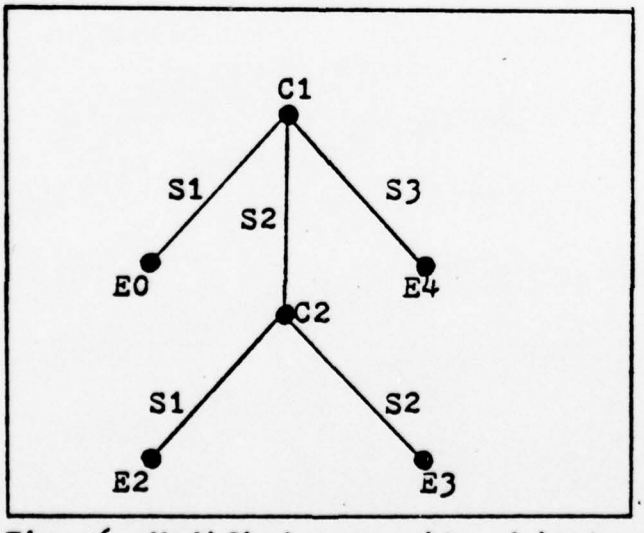


Fig. 6. Modified composite object

Conditional expressions are an example of how predicates can be used. Conditional expressions in the VDL have a format specified in expression 5.

$$P1 \rightarrow E1,\ P2 \rightarrow E2,\ldots,Pn \rightarrow En \qquad (5)$$

where Pi is a predicate, and Ei is an expression defining the action to be taken or a value. The value of a conditional expression is the value of the first Ei for which Pi was true. If no predicate in the conditional expression is true then the value of the conditional expression is undefined. An example of a conditional expression defining the logical "AND" operation is given in expression 6.

$$A \text{ AND } B = (A=0 \rightarrow 0,\ T \rightarrow B) \qquad (6)$$

To obtain a value of "1" for this conditional expression "A" would have to equal "1", and "B" would have to equal "1". If "A = 0" then the value of the expression must be "0". If "A = 0" the first predicate (A=0) is true and the value of the expression is "0". If "A = 1" then the first predicate is false and the second predicate is tested. In this definition the second predicate is defined as always being true. Therefore if "A = 1" the value of the conditional expression is equal to the value of "B". Thus, if "A = 1" and "B = 0", the value of the conditional expression would be "0", if "A = 1" and "B = 1", the value would be "1".

There is another set of operators in the VDL, however, these operators are better classified as elementary functions.

Elementary functions are used in dealing with objects that represent lists. A list is defined to be a string of N objects(none null) selected by <n:0> where $1 \leq n \leq N$. A program is an example of a list in which each element is a statement. Elements in a list are selected by a selector sometimes identified by elem(n). Elementary functions used in VDL are:

LENGTH(L) = total number of elements in the list;

HEAD(L) = object selected from a list when n = 1;

TAIL(L) = a list of objects selected from the original (before this function is executed) list with the original first item(item selected when n = 1) missing. The TAIL(L) will be a list with a length equal to the original length minus one;

$\widehat{L_a\ L_b}$ = concatenation of two lists L_a and L_b to form a single list.

The "L" in the above functions specifies the list currently being operated upon(Ref 3;5;12).

Having covered the Vienna tree, the operatore used to manipulate the Vienna tree, and elementary functions, one is now ready for a brief informal introduction on how a programming language is defined.

The source program, as written by the programmer, is not in the proper form to be interpreted by the VDL interpreter. The VDL interpreter does not operate on program strings, but operates on objects of a Vienna tree. Therefore, the source must be transformed into an abstract form before its semantics can be defined. Transforming the source program is accomplished

by two abstract machines, the ANALYZER, and the TRANSLATOR.

The ANALYZER modifies the programmers source text by verifying that the program text is syntactically(context-free) correct. This is accomplished by comparing the source program text with a context-free syntax specification for the language being defined. A program failing any test is left undefined, that is no attempt is made to determine the meaning, semantics, of the program. A source program with correct context-free syntax is then parsed. The parsed text is represented with selector-object pairs, described on page 35. The parsed text is then processed by the TRANSLATOR.

The TRANSLATOR performs a series of context-sensitive syntax tests on the parsed text supplied by the ANALYZER. The TRANSLATOR also implements some of the affects of certain statements. Types of statements whose affects are implemented by the TRANSLATOR are FORTRAN "explicit type declaration" statements. A FORTRAN program with a statement "REAL I" would cause the TRANSLATOR to locate the variable "I" and change its attributes, causing "I" to be classified as a real variable.

The output of the TRANSLATOR is a parsed text with correct context-free and context-sensitive syntax. This output text is now in the proper format to be processed by the INTERPRETER.

The VDL technique of defining the semantics of a program involves starting in an initial state and changing states until a final state has been reached. This process is handled by the VDL INTERPRETER. The VDL INTERPRETER is based

on the concept of machine states and state transformations.

A state is represented as a composite object(a Vienna tree). The components that compose a state vary with the language being defined. For this introduction a simple state like the state presented in Ref 3 will be described. A state composed of three components, an abstract program, a control object, and a storage object is shown in Fig. 7.

The program object, in Fig. 7, is the parsed text of the abstract program, provided by the TRANSLATOR, and ready to be interpreted. The control object defines which transformations are to be performed. The control object is a composite object whose elementary objects are instructions waiting to be executed. Once an instruction is executed it is erased from the control tree. When all the instructions have been removed from the control tree, the final state has been achieved, and the interpretation is completed. An example of state transformations is given below.

Memory(storage) can be looked upon as three symbol tables,

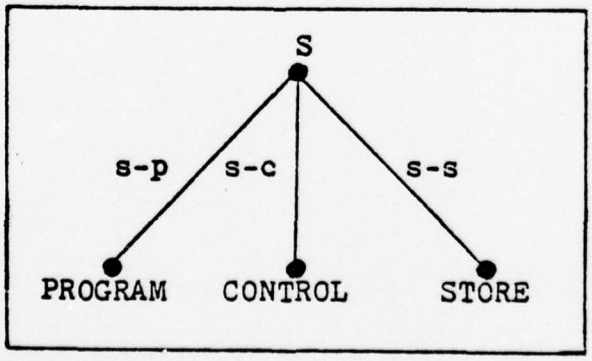


Fig. 7. A Machine State

the environment, the denotation, and the attribute tables(Ref 5). The first table, the environment table, identifies a machine location for every source program identifier. This location contains the value of the identifier. The second table, the denotation table, makes sure that the machine location contains the proper value at all times. The third table, the attribute table, identifies the type(integer, real, etc.) of value identified with each machine location specified in the environment table(Ref 5).

The semantics of a program are described by state transformations. The process of state transformation is performed by the VDL INTERPRETER. The initial state contains a control object with a complete list of the program instructions to be executed. As a program instruction is executed it is removed from the control object thus transforming the state. When the control object is empty the final state has been achieved. These transformations of the control object are accomplished by two types of VDL instructions, self-replacing and value-returning instructions.

Self-replacing instruction are instructions which replace themselves with a subtree of instructions, similar to macro substitutions. An example of a self-replacing instruction is the VDL instruction "VALUE(arg),(when arg is an expression. VALUE(arg) is the name given to a computed value. The commands required to obtain the value of the current source program expression will replace the single instruction VALUE(arg).

The value-returning instruction is the second type of

VDL instruction. The actual semantics of a program, the actual state transformations, are defined by value-returning instructions. Value returning instructions delete the current node being executed from the control tree and pass the computed value to ancestor nodes that reference the just computed value. The VDL instruction "VALUE(arg)" with the arg equal to a constant is a value-returning instruction.

Execution

An exercise to illustrate self-replacing and value-returning instructions and to show the transformation of a control tree, due to the evaluation of one expression "5-3+1", is now given. The exercise begins with one object in the control tree, the VDL instruction "EVAL-EXPR(arg)", shown in Fig. 8 (s-c is the selector name for the control object). EVAL-EXPR(arg) is a self-replacing instruction, therefore, executing EVAL-EXPR(arg) changes the control tree, Fig. 9. The control tree in Fig. 9 has a terminal node of "S:VALUE(A)", this instruction is dependent on its argument as to wether it is a value-returning or self-replacing instruction. The INTERPRETER decides based

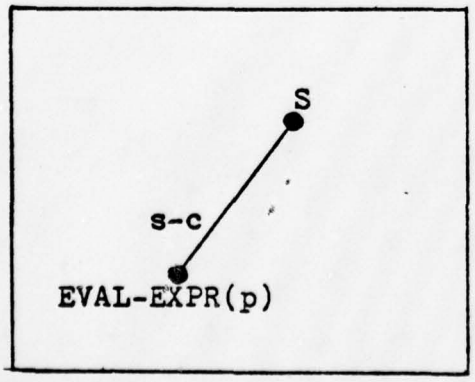


Fig. 8. A control tree.

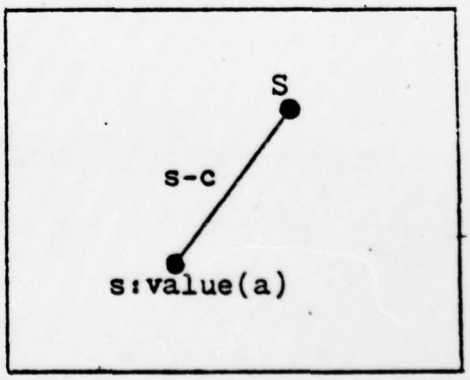


Fig. 9. A control tree

on the following rule.

```
VALUE(arg) = IS-BINARY → APPLY(a,b,S-OP(A)
                              a:VALUE(S1(A))
                              b:VALUE(S2(A))
           IS-CONST → PASS ← A
```

where

```
APPLY(value1,value2,op) = op="+" → PASS ← value1 + value2
                          op="-" → PASS ← value1 - value2
```

An instruction of the form X:VALUE(Y) means the value calculated by VALUE(Y) is assigned to X. The VDL command PASS returns a value to VALUE(A). At present the argument for the "VALUE(A)" instruction is binary(5-3+1), so this instruction is self-replacing, as shown in Fig. 10. Any terminal node on a control tree may be executed for the purpose of illustrating the self-replacing instructions, and then the value-replacing instructions, the left most terminal node, will be

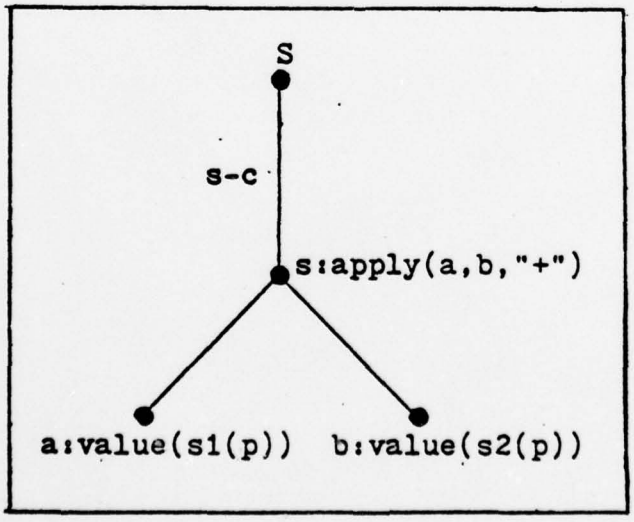


Fig. 10. A control tree

examined next. Executing "a:VALUE(S1(p))" which has a binary argument(5-3) results in another self-replacement that produces the control tree in Fig. 11. Execution of any terminal node in Fig. 11 would demonstrate the value-returning instruction. Executing the left most node "x:VALUE(S1(p))", produces the control tree in Fig. 12. Note the node is removed and the value passed to preceeding nodes that reference "x". To demonstrate the point that any terminal node can be executed, the right most terminal node will be executed next. This is another value-returning instruction, and the result of the execution is shown in Fig. 13. The control tree is Fig. 13 has only one terminal node to execute. Execution of this node leads to the control tree in Fig. 14. Execution of the terminal node in Fig. 14 leads to the control tree in Fig. 15, and execution of the terminal

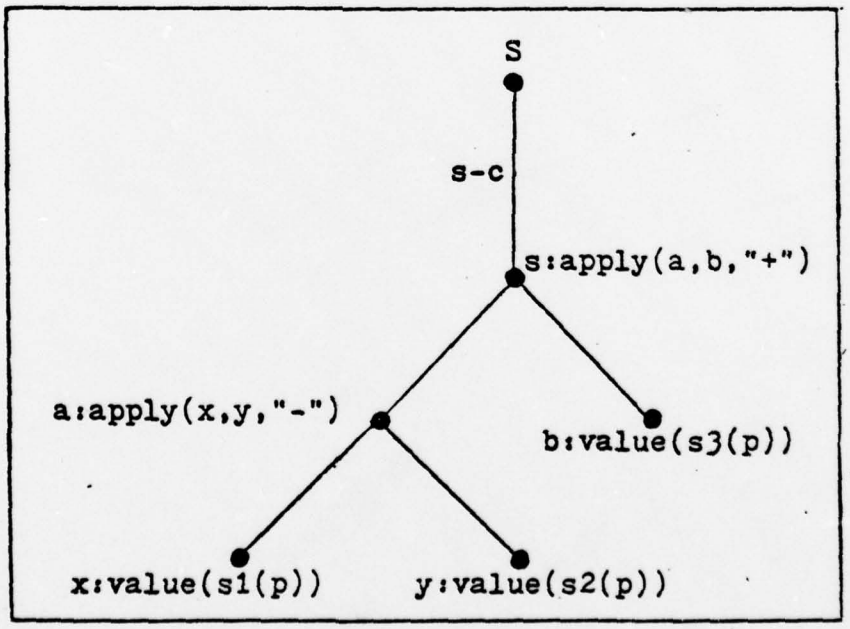


Fig. 11. A control tree

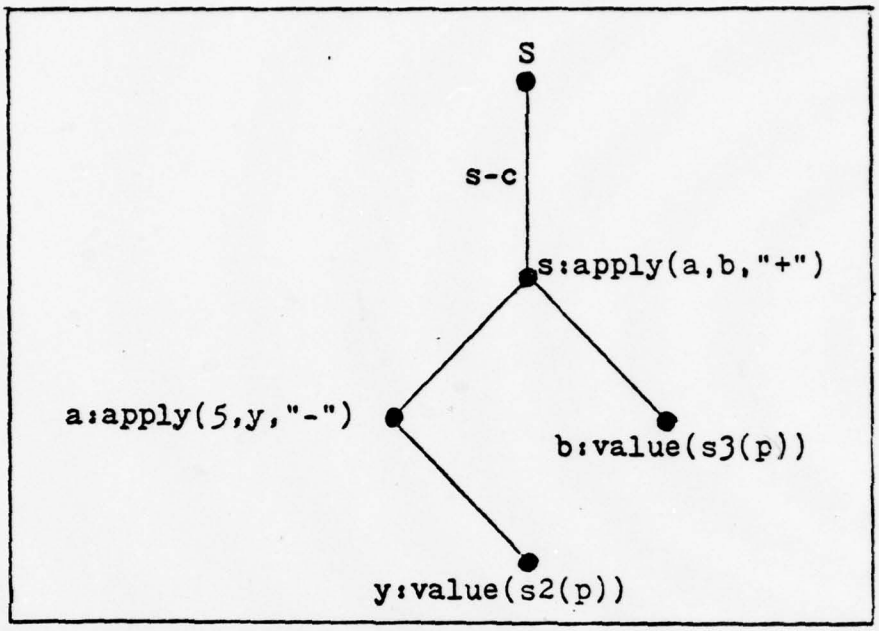


Fig. 12. A control tree

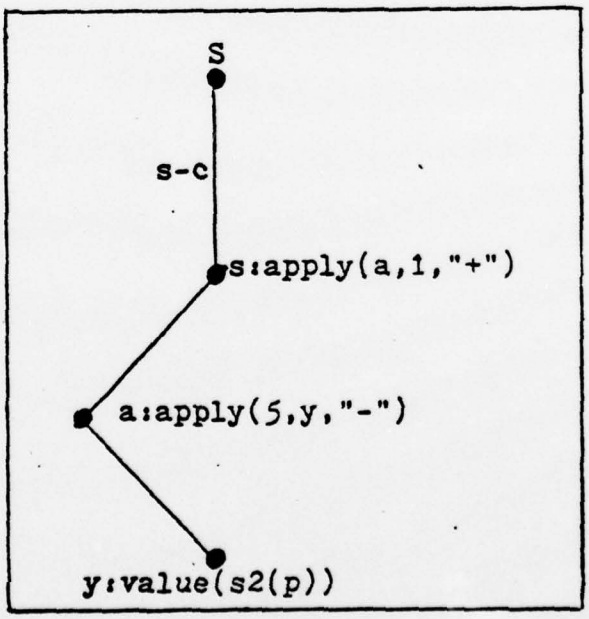


Fig. 13. A control tree

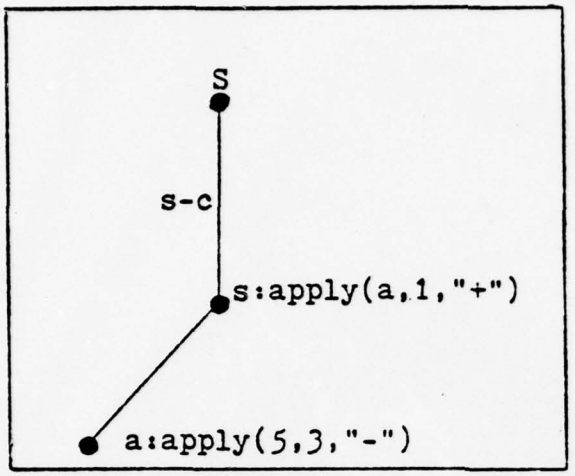


Fig. 14. A control tree

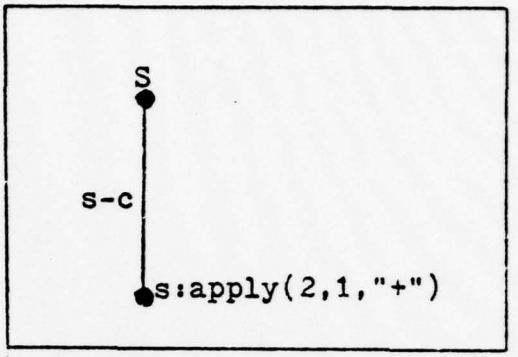


Fig. 15. A control tree

node in the control tree in Fig. 15 calculates the sum of "2 + 1" and produces an empty control tree. An empty control tree signifies the final state, thereby signifying a successful statement evaluation.

The above exercise demonstrated how state transformations occur since each different control tree represents a seperate state. This exercise also illustrated how conditional statements are used in defining the VDL abstract machines, which will be illustrated again in the example at the end of this chapter.

The relationship between each of the three abstract machines is illustrated in Fig. 16. This completes the examination of the individual components of a VDL definition.

Sewing all the components of a VDL definition together is accomplished by a definition of the same program and language ASL presented in chapter IV. The language consists of

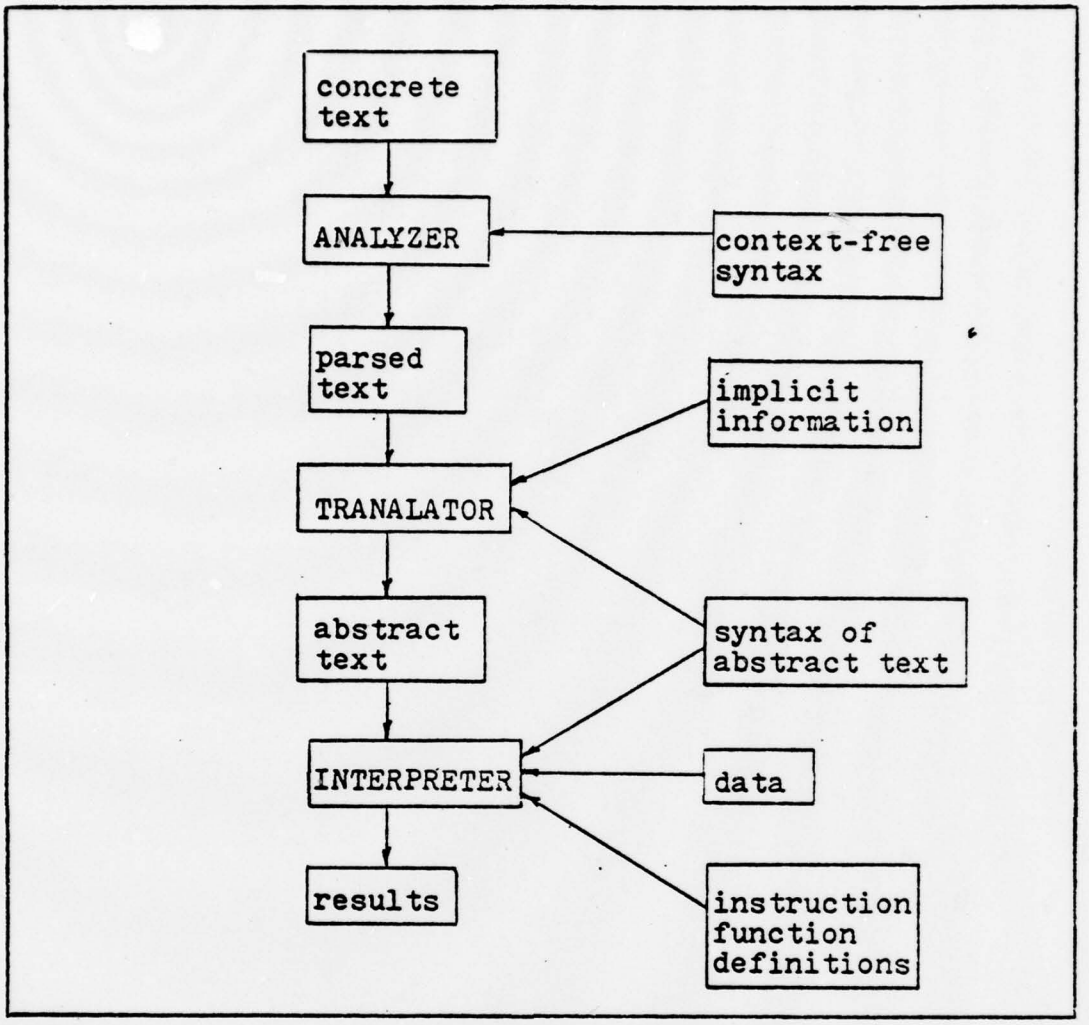


Fig. 16. The language definition machine (Ref 12,Fig. 5.6)

assignment statements, a stop statement, and an end statement. Each statement except the end statement is terminated with a semi-colon. The format of the assignment statement consists of "LET" followed by a single alphabetic character(A, B, or C) then "=" followed by an unsigned integer constant(LET A=3). The test program is

```
LET A = 1;
LET B = 2;
END
```

The ANALYZER is the first machine to process the test program. The ANALYZER checks the context-free syntax of the test program according to the following rules.

IS-PROGRAM = (S1:IS-STMT-LIST, S2:IS-END),

IS-STMT-LIST = (<S-DEL:IS-;>, <S1:IS-STMT>,...),

IS-STMT = IS-ASMT-STMT v IS-STOP,

IS-ASMT-STMT = S1:IS-LET, S2:IS-LETTER, S3:IS-=,
S4:IS-INT-CONST,

IS-LETTER = IS-A v IS-B v IS-C,

IS-INT-CONST = IS-1 v IS-2 v IS-3 v IS-4 v IS-5 v IS-6 v
IS-7 v IS-8 v IS-9 v IS-0,

where (<S-DEL:IS-;>,<S1:IS-STMT>) signifies that each statement terminates with a semi-colon.

With the test program as input to the ANALYZER the output of the ANALYZER, a parsed text, is illustrated in Fig. 17.

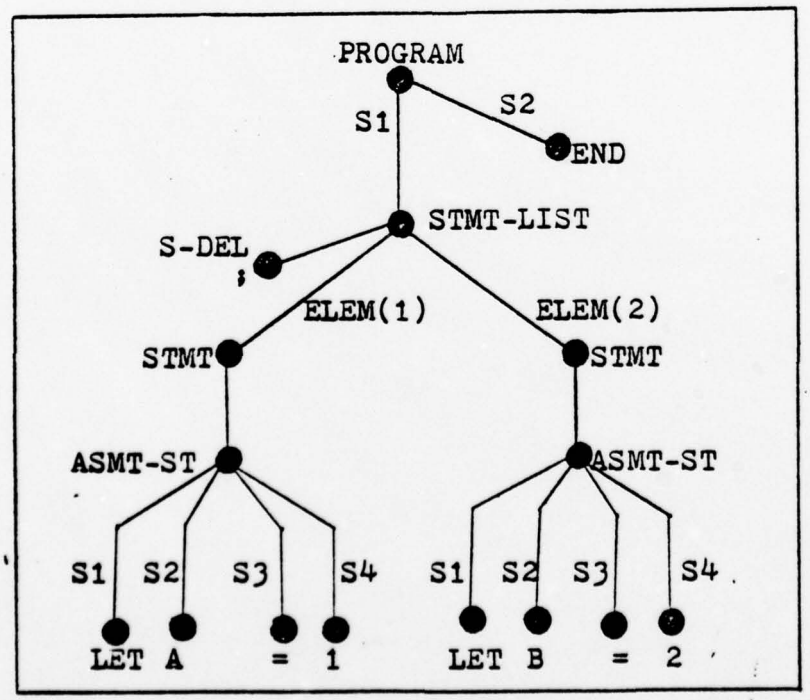


Fig. 17. The ANALYZER output.

The parsed text from the ANALYZER is then passed to the TRANSLATOR for context-sensitive testing. The set of conditional statements defining the TRANSLATOR for the test language is given below with comments in quotation marks. The argument "p" represents the parsed program text inputed to the TRANSLATOR.

```
TRANSLATE(p) = program-length(p) ≤ max-program-length →
                    TRANS-STMT-LIST(p)
                    TRUE → ERROR "program too long"
TRANS-STMT-LIST(p) = LENGTH(p) = 0 → <>  "if there are
no statements length(p) = 0 returns empty set <>"
                        TRUE → u_o({<elem(i):TRANSLATE-ASSIGN-
                                MENT(Si(p))> || 1 ≤ LENGTH(p)})
TRANSLATE-ASSIGNMENT(p) = TRUE → u_o(<target:MAKE-ID(S1(p))>,
                                <source:MAKE-INT-CONST(p)>)
MAKE-ID(p) = length of ID ≤ max-ID-length → LETTER(p)
                TRUE → ERROR  "ID is too long"
LETTER(p) = IS-A(p) → A
            IS-B(p) → B
            IS-C(p) → C
MAKE-INT-CONST(p) = VALUE-INT-CONST(p) ≤ max-value →
                    VALUE-INT-CONST(p)
                    TRUE → ERROR  "integer constant too big"
VALUE-OF-INT-CONST(p) = IS-1(p) → 1
                        IS-2(p) → 2
                        IS-3(p) → 3
                        IS-4(p) → 4
```

IS-5(p) → 5

IS-6(p) → 6

IS-7(p) → 7

IS-8(p) → 8

IS-9(p) → 9

IS-0(p) → 0

The implementation restriction listed above max-program-length, max-ID-length, and max-value are user defined. The output of the TRANSLATOR is an abstract parsed program text, illustrated in Fig. 18.

The abstract program text produced by the TRANSLATOR is now ready to be interpreted, (have its semantics defined). The interpretation process involves changing states through changes in the control tree, as shown earlier. The argument "p" in the rules which define the INTERPRETER represents the abstract program manipulated by the INTERPRETER. The rules that define the INTERPRETER for the test language are:

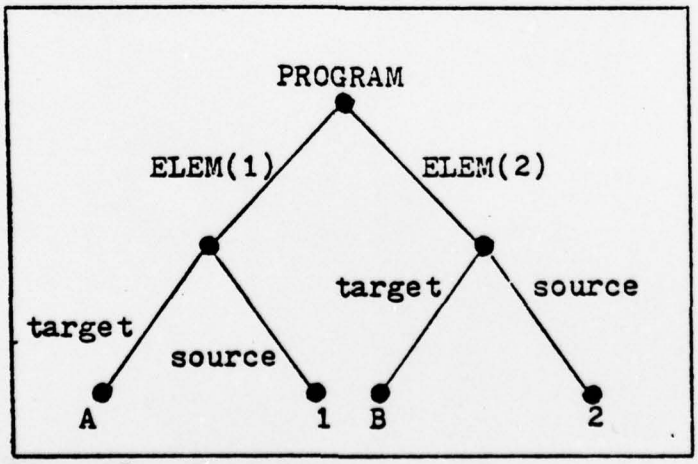


Fig. 18. Translator output

```
INTERPRET-STATEMENT-LIST(p) = IS-    → NULL   "if no statements
                                              do nothing"
                              TRUE → INTERPRET-STATEMENT-LIST(TAIL(p));
                                     INTERPRET-STATEMENT(HEAD(p))
```

"look at the control object as a LIFO stack, the head of the program(the first statement) is the first instruction to be processed."

```
     INTERPRET-STATEMENT(p) = IS-ASMT(p) → INTERPRET-ASSIGNMENT(p)
    INTERPRET-ASSIGNMENT(p) = ASSIGN(target(p), VALUE);
                              VALUE: EVAL-EXPR(source(p))
               EVAL-EXPR(p) = IS-VALUE(p) → PASS:p
      ASSIGN(target,VALUE) = u(store(state-s): target:VALUE)
```

The above defined INTERPRETER is ready to process the abstract program provided by the TRANSLATOR. The INTERPRETER executes the statements in the control object beginning in an initial state and finishing in a final state(an empty control object).

The initial state for the test program is shown in Fig. 19 In Fig. 19 the program is the abstract program, the control object and the storage object are predefined for the initial state. Rather than redraw the entire state for each state transition only the control tree will be shown until the source program is completely interpreted, then the entire state will be shown. The terminal node of the control tree in the initial state is a self-replacing instruction "INTERPRET-STATEMENT-LIST", the execution of which leads to the next state, Fig. 20.

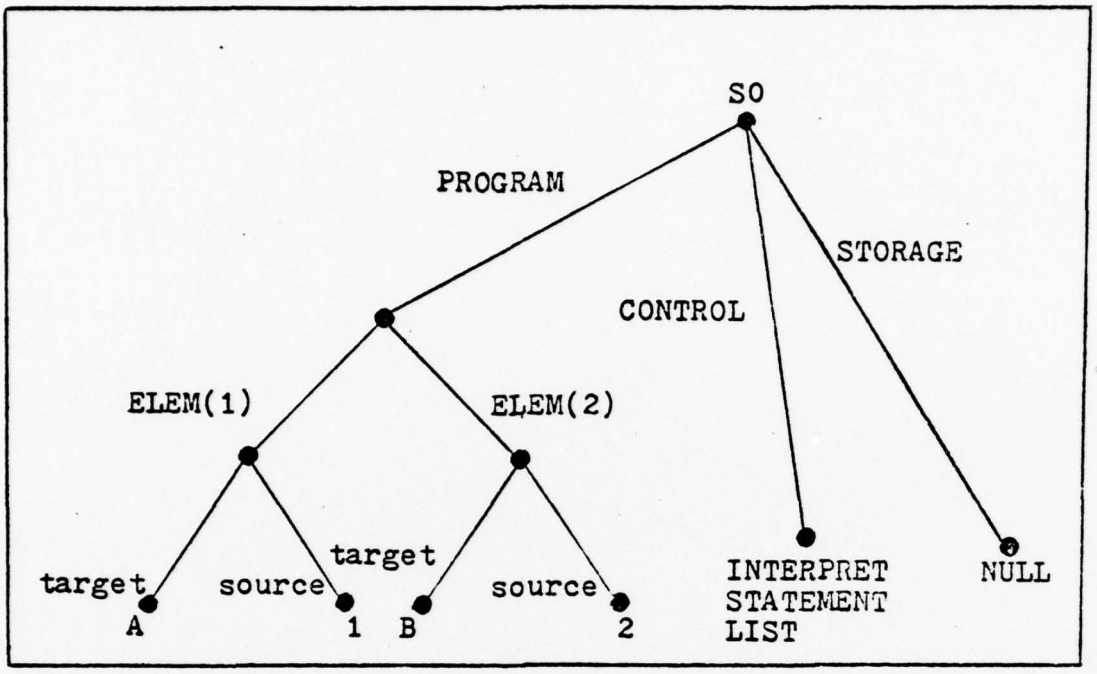


Fig. 19. The initial state.

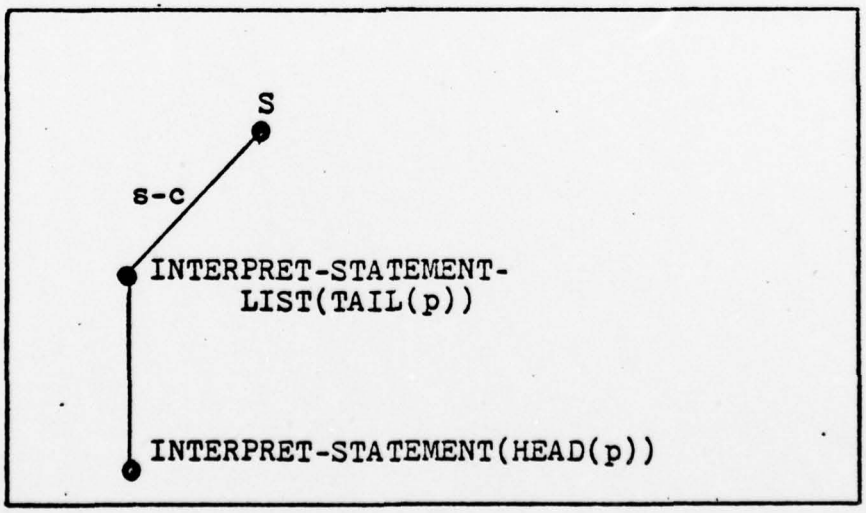


Fig. 20 The control object for the first state

Execution of the terminal node of the control tree of the first state, Fig. 20, leads to the second state shown in Fig. 21. The terminal node of the control tree for the second state contains another self-replacing instruction, the interpretation of which leads to state three in Fig. 22. The terminal node of the control tree for the third state contains a value-replacing instruction, whose execution leads to the fourth state, Fig. 23. Execution of the terminal node in Fig. 23 leads to state five, Fig.24, and the storage has changed from NULL to the storage shown in Fig. 25. Executing the control object in state five will initiate the same process again because the next statement is also an assignment statement. The final state of the interpretation is shown in Fig. 26. Note that in the final state the control object is always NULL.

This chapter has presented a brief informal description of the Vienna Definition Language, and has presented an example which shows how a test program is processed by a VDL language definition.

With a brief introduction to the VDL complete, the process of examining BASIS/1-12 can begin. The following chapter, chapter VI, provides a brief introduction to IBM's BASIS/1-12 technique.

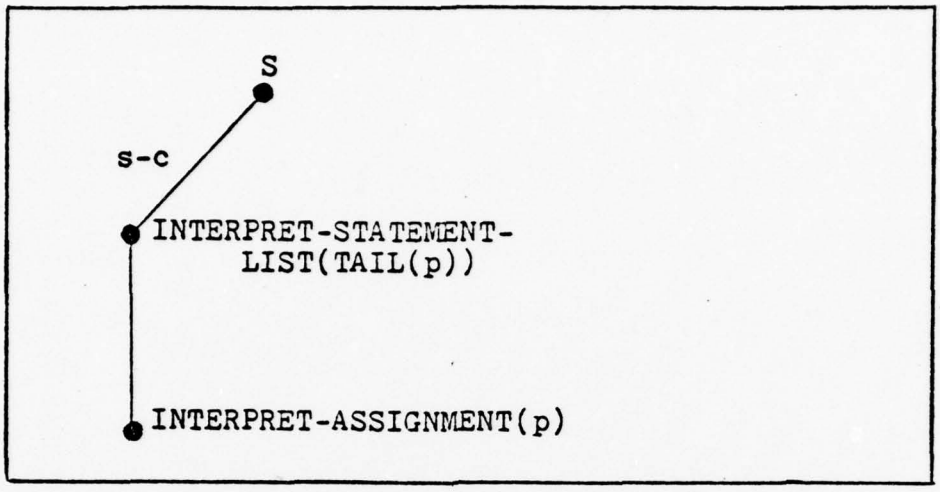


Fig. 21. The control object for the second state

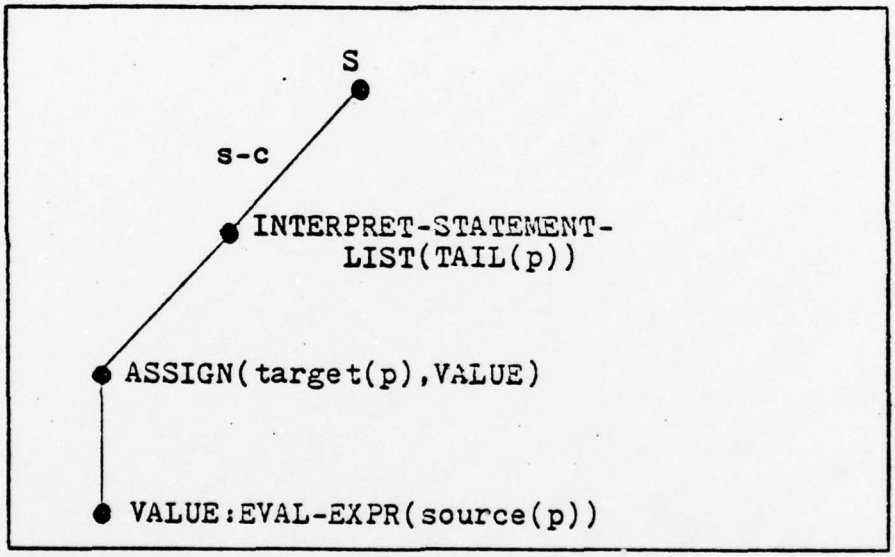


Fig. 22. The control object for the third state

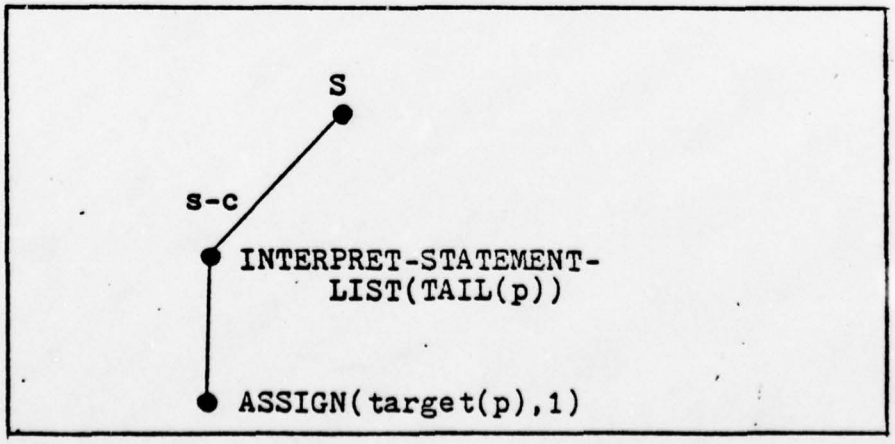


Fig. 23. The control object for the fourth state

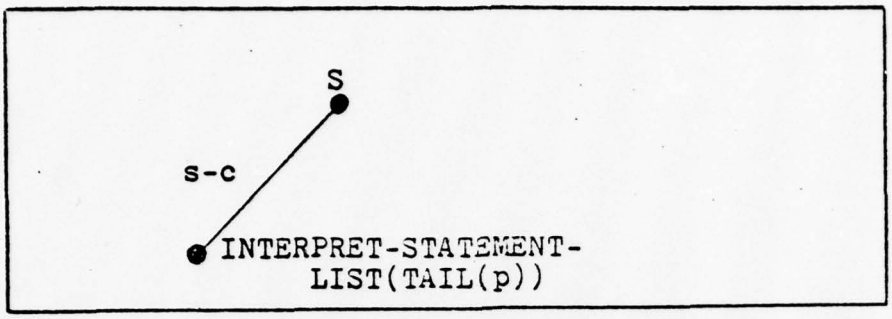


Fig. 24. The control object for the fifth state

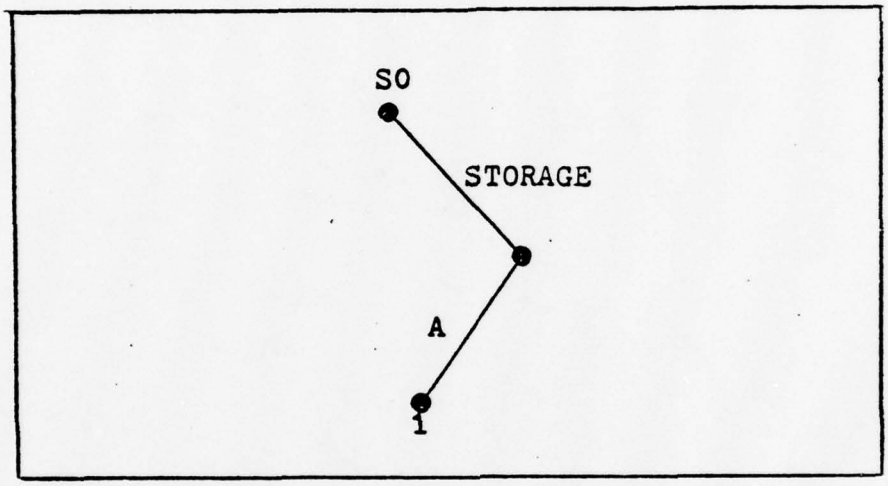


Fig. 25. The storage object for the fifth state

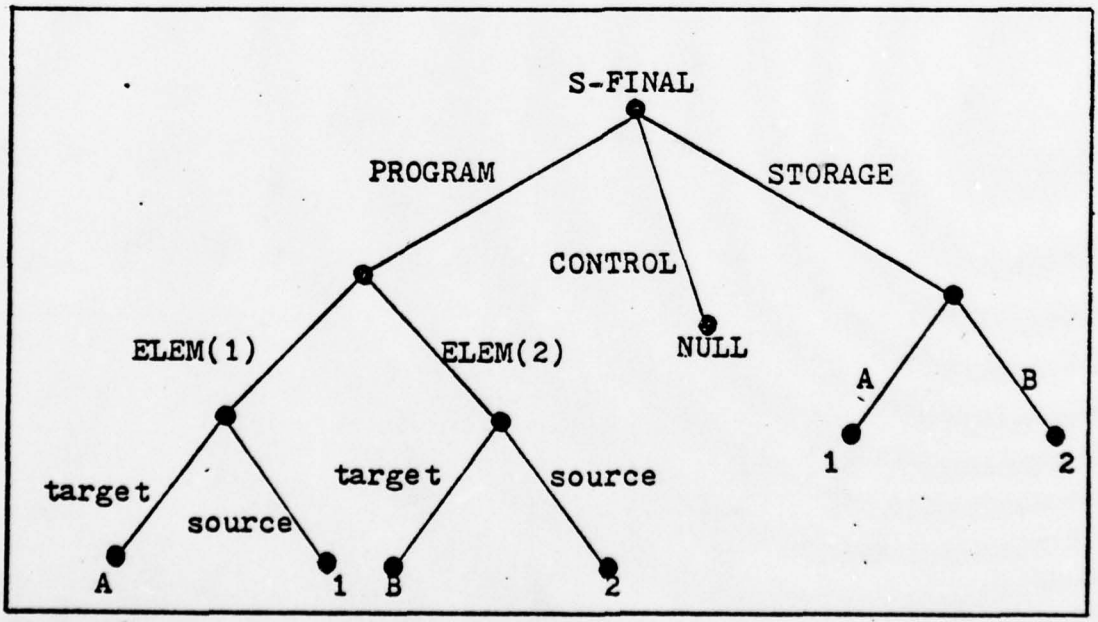


Fig. 26. The final state

VI BASIS/1-12

Background

Research conducted at IBM laboratories in Hursley, England resulted in a semi-formal language definition technique known as BASIS/1-12. Personnel at the Hursley laboratories prefered a semi-formal approach for language definition techniques rather than the formal approach persued by the Vienna laboratories in their development of the Vienna Definition Language(VDL). BASIS/1-12, though semi-formal, is very similar to VDL in that the same concepts of abstract machines, state transformations, and an abstract representation of the source program are used. BASIS/1-12 is semi-formal because state transformations are defined with operations written in English prose. On August 9, 1976, the BASIS/1-12 definition of PL/I was used by the American National Standards Institute (ANSI) in establishing a national standard for the programming language.

This chapter contains a brief look at the BASIS/1-12 technique, consisting of a brief summary of "The Definition Mechanism For Standard PL/I" by Marcotty and Sayward(Ref 16) and an example of the technique defining the assignment statement language(ASL), presented in chapter IV.

Methodology

The BASIS/1-12 technique of defining a programming language appears to be a combination of SEMANOL, presented in

chapter IV, and the Vienna Definition Language(VDL), presented in chapter V.

The concept of machine state and state transformations are utilized, as in the VDL, and the semantics are defined with algorithms, as in SEMANOL. However the algorithms in BASIS/1-12 are written in English. Therefore BASIS/1-12 appears to be more closely related to the VDL. The basic machine state of BASIS/1-12 and the VDL are identical, each containing information that controls the machines operations, the program being defined, and the values of variables and data required by the source program being defined. The BASIS/1-12 machine state, though represented as a tree, is not represented as a "Vienna tree", i.e. the branches are not labeled.

The Meta-language

The details of the BASIS/1-12 technique will be covered in the following order: the structure of the BASIS/1-12 machine state; the instructions used to manipulate the machine state; the BASIS/1-12 abstract machines; and the structure of the algorithms used to "operate" the machine.

The structure of the data manipulated by the BASIS/1-12 machine is that of a tree. The "BASIS/1-12 tree" has a unique feature in which each node of the tree has a unique name, allowing each node to be referenced. This feature is illustrated in Fig. 27. The node contents of Fig. 27 are represented by alphabetic characters, and the unique node names are represented by digits. While the contents of several nodes may

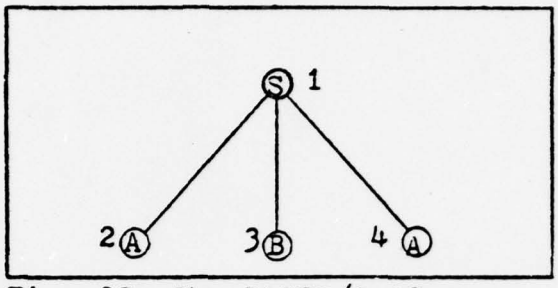


Fig. 27. The BASIS/1-12 tree

be identical a node's name is unique. Nodes "2" and "4" have the same content in Fig. 27. A term assiciated with the tree structure is "the son of a node". In Fig. 27 a son of node "1" is either node "2", "3", or "4", that is the son of a node is an immediate component of that node.

Throughout the definition the source program is maintained as part of the machine state. The source program is in the form of a tree with the appropriate syntax rules governing the structure of the tree. During the definition the source program is first a "character" program in which all the characters of the source program form the terminal nodes of the tree. The "character" program is then transformed into an "abstract" program in which only abstract objects form the terminal nodes of a tree.

Another feature of the "BASIS/1-12 tree" is the "designator node". Related to the unique name concept, the designator node is a node which refers or points to another node. An example of a designator node is the <TYPE-DESIGNATOR> node in Fig. 28. The arrow in Fig. 28 is added to demonstrate the meaning of a designator node.

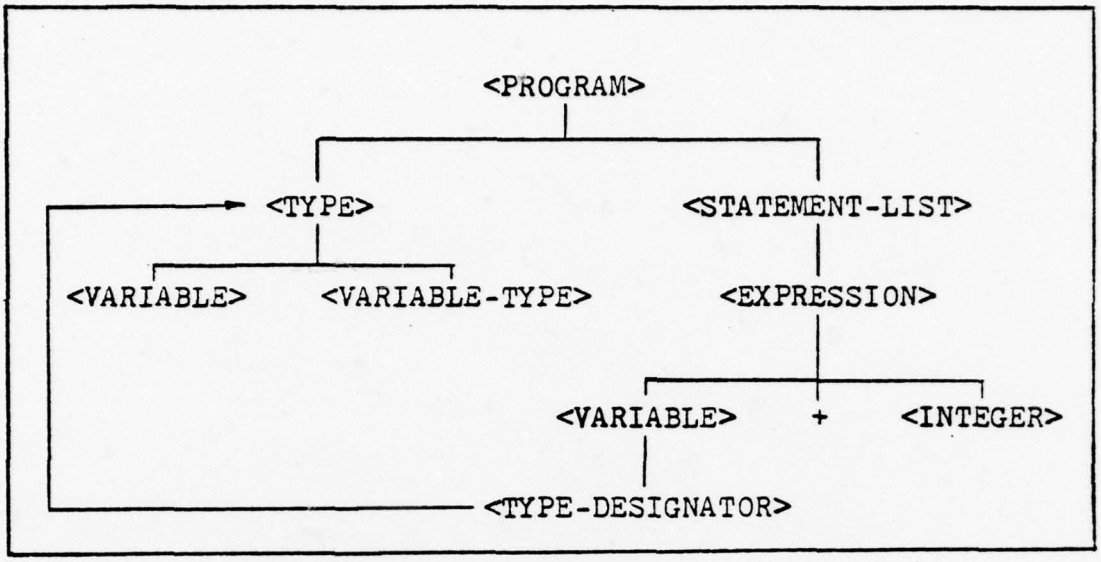


Fig. 28. The BASIS/1-12 designator node

Instead of drawing out a tree every time a tree structure is being refered to, an indentation notation is used. The indentation format involves listing a root node of a tree and below it, indented equally, the sons of the root node. For example, the tree in Fig. 31 (page 63) would be represented as

```
<Z>
  <S>
    <A>
    <B>
    <A>
  <B>
    <X>
```

The indentation notation proves to be very useful when specifying new tree structures.

The basic structure of the machine state, the BASIS/1-12 tree, has been described above, now the instruction used to manipulate this structure will be defined.

Execution

The syntax and semantics of a source program are defined by manipulating the machine state. The instructions used to manipulate the tree are used directly in the algorithm. These manipulation instructions fall into two basic catagories: those whose effects depend on the tree structure(syntax) of the defined program and those whose effect do no depend on the syntax of the defined program.

There are two instructions that depend on the syntax of the defined program: the ATTACH and DELETE instructions. The ATTACH instruction attaches a specified tree to a specified node, inserting the proper nodes to make the attachment syntactically valid. Using the two syntax rules below:

<Z> ::= A | A<B>

<B> ::= X | Y

the command

ATTACH X to N

where N is the unique name of the node whose content is Z in Fig. 29. Results in the tree structure presented in Fig. 30. Node N2 in Fig. 30 was inserted to make the structure of the tree(the syntax) valid.

The other instruction whose effect is dependent on the structure of the tree is the DELETE instruction. The DELETE

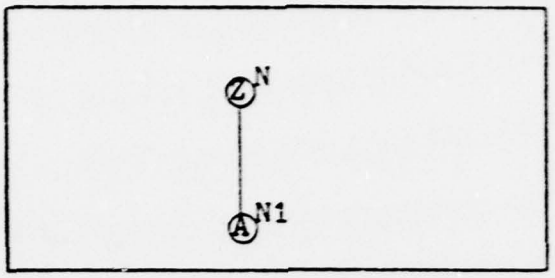


Fig. 29. The original node N

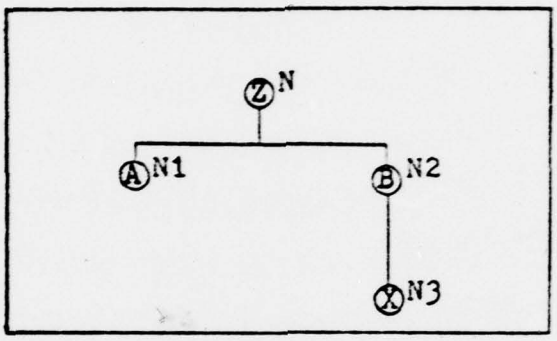


Fig. 30. The redefined node N

instruction causes the designated node and its associated tree (the designated node is the root node of its associated tree) to be deleted. If the tree remaining after the specified deletions have been made is not valid then other nodes, as required to make the tree valid are deleted.

The effects of the remaining three instructions, LET, REPLACE, and APPEND do not depend on the tree structure.

The LET instruction makes a local variable designate an existing tree. The LET instruction below

LET L be

<Z>

<A>

<B>

<X>

specifies the local variable "L" as designating the entire tree in Fig. 30.

The REPLACE instruction replaces a specific tree with another tree designated by a local variable. The format of the REPLACE statement is

REPLACE A designated by N1 by a copy of 1

Execution of this statement using the trees in Fig. 27 and Fig. 30 results in the tree in Fig. 31. Note that the unique name of the root node of the tree to be replaced now becomes the unique name of the root node of the new tree.

The last tree manipulation instruction is the APPEND instruction. "The APPEND instruction attaches a tree to the rightmost element of a list."

The above five commands are used throughout the BASIS/1-12 definition to manipulate the machine state into reflecting changes caused by the "execution" of the source program. The framework from which the tree structure is manipulated, the abstract machine, is defined below.

BASIS/1-12 is an interpreter-oriented technique utilizing

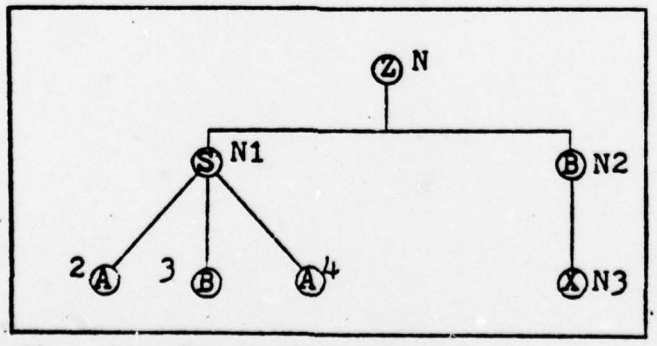


Fig. 31. The modified node N

an abstract machine. The abstract machine can be defined as the combination of three abstract machines: an ANALYZER, a TRANSLATOR, and an INTERPRETER.

The ANALYZER takes the string representation of the source program, removes the blanks from the program, and parses the program according to the syntax rules of the defined language. The output of the ANALYZER is the parsed source program in tree form, with the characters of the source program as the terminal nodes of the tree.

The TRANSLATOR takes the output of the ANALYZER, performs context-sensitive tests, removes any characters associated with the string structure of the program(e.g. parenthesis, semi-colons, etc.), and then outputs an abstract program, a simplified version of the source program. The abstract pro- is also represented by a tree structure.

The final process involves the INTERPRETER. The INTERPRETER "executes" the abstract program, changing machine states as it responds to the syntax and semantic definitions being utilized. The output of the INTERPRETER is the set of machine states generated and the output file generated by the "exection" of the source program.

The interface between each machine is shown in Fig. 32.

When it is not important to distinguish between these machines they will be considered as one machine.

With the BASIS/1-12 machine defined, the operations performed in defining a source program can now be defined. Only the operations utilized in the example at the end of this

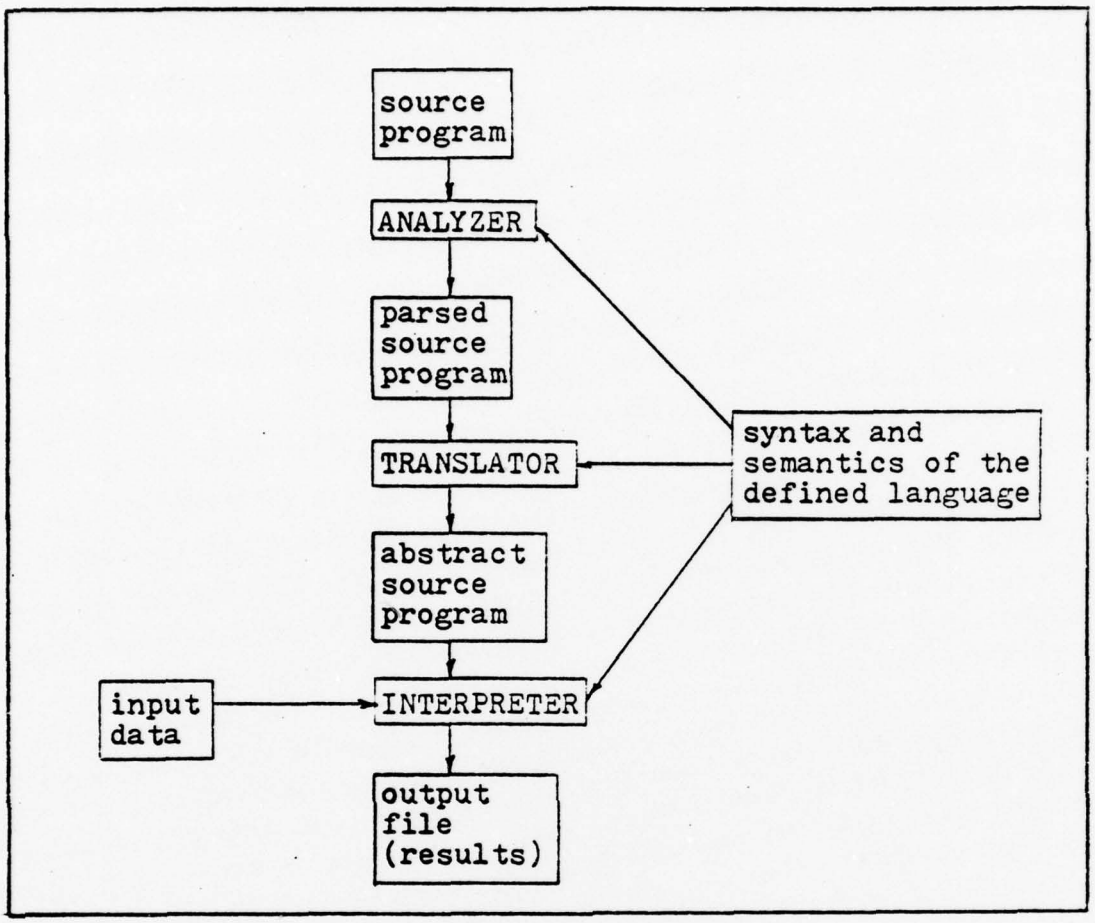


Fig. 32. The language definition machine

chapter will be described.

The BASIS/1-12 machine state is manipulated and thus transformed by a set of operations. These operations use a small set of data manipulating instructions, enabling one to reference specific nodes on a tree, construct new temporary trees, manipulate trees, do arithmetic, and establish local variables. Operations also have the capability of calling other operations. Calling an operation causes that operation to be placed at the rightmost node of the operations tree; thus, making it the current operation. The operation in

control of the BASIS/1-12 machine is the rightmost operation of the operations tree, if there are two operations trees the rightmost tree is executed first. Upon completion an operation is deleted from the operations tree, and control returned to the new rightmost operation.

An operation always begins with the word "OPERATION" followed by the name of the operation. The body of the operation consists of steps and cases. Steps in the operation are executed either sequentially, conditionally, or iteratively, while the cases are mutually exclusive. An operation consisting of cases must have at least one case satisfied, or the operation is undefined(similar to the conditional statements in the VDL).

Instructions(steps) in the operations are executed sequentially, however, there are instructions that alter the flow of control. Control altering instructions presented are: GO TO, FOR EACH, PERFORM, and RETURN instructions.

The GO TO instruction transfers control to another step within the operation. The FOR EACH instruction causes a series of instructions to be executed once for every item specified in the FOR EACH statement. The PERFORM instruction "calls" another operation, thereby, transfering control to the called operation. The RETURN instruction returns control to the operation that called the current operation. Control is transfered to the step following the PERFORM instruction in the calling operation.

The operations used in BASIS/1-12 are straightforward,

easy to follow, and several operations are demonstrated at the end of this chapter.

An example of how to follow the flow of control between operations is given below. The two sample operations are:

```
OPERATION:  OP1
STEP 1.  PERFORM OP2
STEP 2.  PERFORM OP3
   .
   .
   .
OPERATION:  OP2
   .
   .
   .
```

The first step of operation OP1, "PERFORMOP2", transfers control to OP2. Therefore, the reader should "leave" OP1 at step 1 and execute OP2. After OP2 is executed the reader should return to OP1 at the step after the respective "PERFORM" step, in this case the reader returns to step 2.

Along with the flow of control the reader must also keep track of local variables. The basic purpose of local variables is demonstrated by the ANALYZE-PARSE operation. Step 1 of the ANALYZE-PARSE operation is "Obtain a source program the structure of which is a character list, cl." At this point the source program is assigned the name cl. When a local variable is used in parenthesis it is used as a parameter. Step 2 of the ANALYZE-PARSE operation is "PERFORM PARSE(cl) to get program, cp." In step 2 "cl" is used as a parameter for the PARSE operation, that is, the PARSE operation will operate on cl. The result of the PARSE operation, a parsed

program, is then given the name cp. Referencing the parsed program is accomplished by using the local variable cp. The reader is now prepared to follow the example below.

The following is a very simple example of a BASIS/1-12 language definition for the test language ASL presented in chapter IV. The test language consists of assignment statements, a stop statement, and an end statement. Each statement except the end statement is terminated with a semi-colon. The format of the assignment statement consists of "LET" followed by a single alphabetic character(A, B, or C) then "=" followed by an unsigned integer constant(LET A = 3). The test program is

```
LET A = 1;
LET B = 2;
END
```

The initial state of the BASIS/1-12 machine is illustrated in Fig. 33. The trees illustrated in this example will have only the unique names of nodes which are referenced. It is important to remember that every node has a unique name.

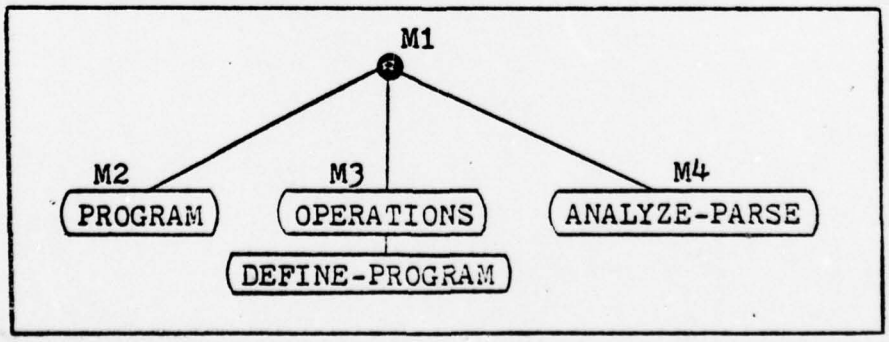


Fig. 33. The initial state

The BASIS/1-12 machine is controlled by the rightmost node of the operations tree. Examing the initial state(Fig. 33) reveals that the rightmost node of the operations tree is "DEFINE PROGRAM". Therefore the operation DEFINE PROGRAM will be executed next. The DEFINE PROGRAM operation consists of three steps

OPERATION: DEFINE PROGRAM

STEP 1. PERFORM ANALYZE-PARSE

STEP 2. PERFORM TRANSLATE-PARSE

STEP 3. PERFORM INTERPRETATION

The execution of step 1 causes the ANALYZE-PARSE operation to become the rightmost node of the operations tree and thus in control. The effect of step 1 is illustrated in Fig. 34. Before describing the ANALYZE-PARSE operation which performs the initial parse on the source program, the syntax rules used for the parse will be given. The context-free syntax checks

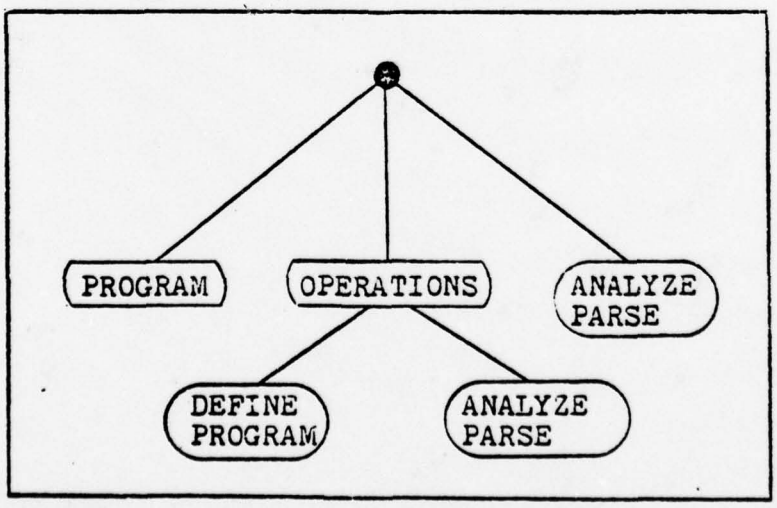


Fig. 34. Step 1 of DEFINE-PROGRAM

performed by the ANALYZE-PARSE operation are divided into two levels, low and high level checks. The low level check verifies that the characters in the source program are either words or constants. The high level check verifies that the words and constants make up a valid program. The low level syntax rules for ASL are:

<TEXT> ::= <DELIMIT-PAIR><DELIMIT-PAIR>

<DELIMIT-PAIR> ::= <WORD><DELIMITER>

<DELIMITER> ::= ∅ | = | ;

<WORD> ::= <LETTER> | <DIGIT> | LET | STOP | END

<LETTER> ::= A | B | C

<DIGIT> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0

The meta-symbol [] means zero or more repetitions of the contents, while ∅ represents a blank. The high level syntax rules for ASL are:

<PROGRAM> ::= <STATEMENT-LIST>[STOP] END

<STATEMENT-LIST> ::= <STATEMENT-LIST><STATEMENT> ;

<STATEMENT> ::= <ASSIGNMENT-STATEMENT> | STOP

<ASSIGNMENT-STATEMENT> ::= <LETTER> = <DIGIT>

The two level syntax checks make it easier to remove the blanks from the source language.

The operation currently in control is the ANALYZE-PARSE operation. The operations ANALYZE-PARSE and PARSE are given below.

OPERATION: ANALYZE-PARSE

STEP 1. Obtain a source program the structure of which is a character list,cl.

STEP 2. PERFORM PARSE(cl) to get program, cp.

STEP 3. ATTACH cp to <ANALYZE-PARSE>.

OPERATION: PARSE(cl)

where: cl is a character list

result: a program

STEP 1. PERFORM low level parse(cl) to get a text, tx.

STEP 2. PERFORM high level parse(tx) to get a program, cp.

STEP 3. RETURN.

The two operations high and low level parse are not listed because their effects are obvious. The result of the ANALYZE-PARSE operation is a program with the blanks removed. The result of ANALYZE-PARSE on the test program is illustrated in Fig. 35. With its completion the ANALYZE-PARSE operation deletes itself from the operations tree and returns control to DEFINE-PROGRAM. The current location within DEFINE-PROGRAM is step 2; therefore, step 2 is executed. Step 2 of DEFINE-PROGRAM is PERFORM TRANSLATE-PARSE, which causes the operation TRANSLATE-PARSE to become the rightmost node of the operations tree and therefore in control. The operation TRANSLATE-PARSE conducts context-sensitive tests(not required for ASL), constructs the abstract program, and changes the machine state to prepare for the interpretation phase. Because context-sensitive test are not required for ASL the TRANSLATE-PARSE operation consists of only two steps, listed below. Step 1 creates the abstract program while step 2 prepares the machine state for the inter-phase.

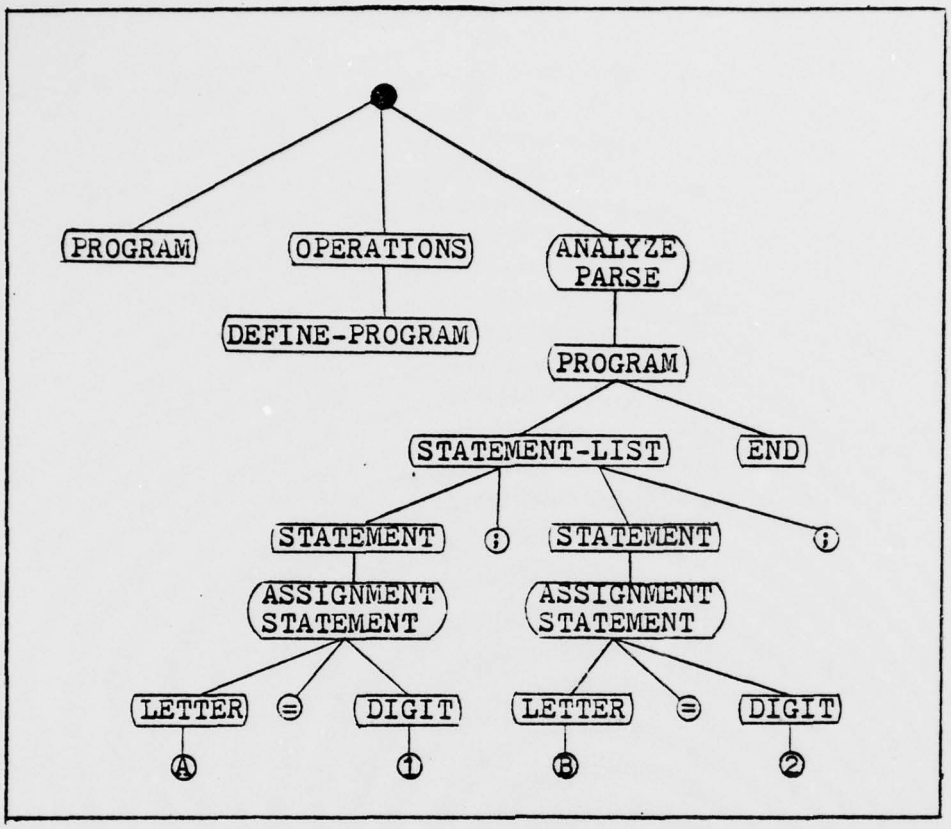


Fig. 35. The result of executing ANALYZE-PARSE

OPERATION: TRANSLATE-PHASE

STEP 1. PERFORM CREATE-ABSTRACT-PROGRAM.

STEP 2. DELETE the ANALYZE-PARSE from the machine state.

The CREATE-ABSTRACT-PROGRAM operation consists of one step and is self-explaining.

OPERATION: CREATE-ABSTRACT-PROGRAM

STEP 1. FOR EACH STATEMENT, st

PERFORM CREATE-ABSTRACT-STATEMENT(st).

The CREATE-ABSTRACT-STATEMENT operation utilizes the program illustrated in Fig. 35. The results of the CREATE-ABSTRACT-STATEMENT operation and step 2 of the TRANSLATE-PARSE operation

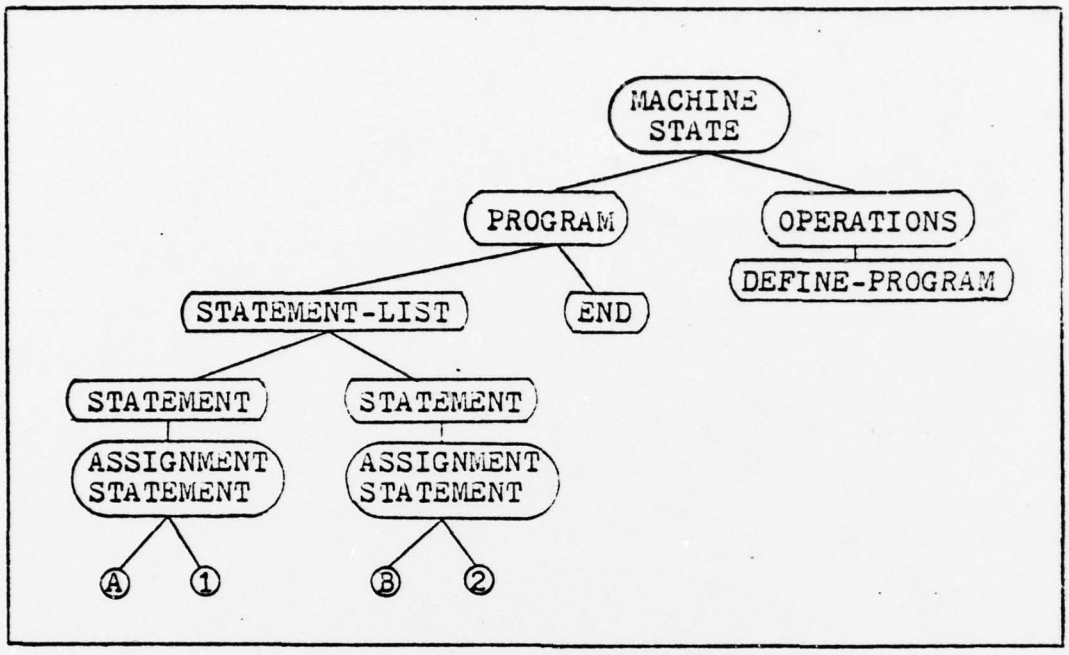


Fig. 36. The abstract assignment statements

is shown in Fig. 36. The CREATE-ABSTRACT-STATEMENT operation removes the "LET" and the "=" from the assignment statements. Note that the new assignment-statement tree in Fig. 36 was specified in the indentation format in step 2 of the CREATE-ABSTRACT-STATEMENT operation listed below.

OPERATION: CREATE-ABSTRACT-STATEMENT(cas)

where: cas is an assignment-statement

result: an abstract assignment-statement

STEP 1. LET L and D, be respectively, the sons of <LETTER> and <DIGIT> of cas

STEP 2. RETURN an <assignment-statement>

<LETTER>

<L>

<D>

Upon completion of the TRANSLATE-PARSE operation the rightmost node of the operations tree is again DEFINE-PROGRAM. Step 3 of the DEFINE-PROGRAM operation is now in control. The general format of the machine state during the interpreteation phase is illustrated in Fig. 37.

The first operation executed in the interpretation phase of the language definition is the INTERPRETATION operation listed below.

OPERATION: INTERPRETATION

STEP 1. LET std be a statement designator that specifies the first element of statement-list of <PROGRAM>.

"In Fig.37 the STATEMENT-DESIGNATOR points to the node with the unique name of "1".

STEP 2. ATTACH to INTERPRETATION the tree

<PROGRAM-STATE>

<STORAGE>

<DIRECTORY>

<CELLS>

<PROGRAM-CONTROL>

<std>

STEP 3. APPEND an operation for program execution to <PROGRAM-CONTROL>

"Steps 2 and 3 added the entire PROGRAM-CONTROL tree shown in Fig. 37. Note that the rightmost operation node is now the PROGRAM-EXECUTION node in the PROGRAM-CONTROL tree. The operations in the PROGRAM-CONTROL tree will remain in control until the interpretation is complete, then control transfers

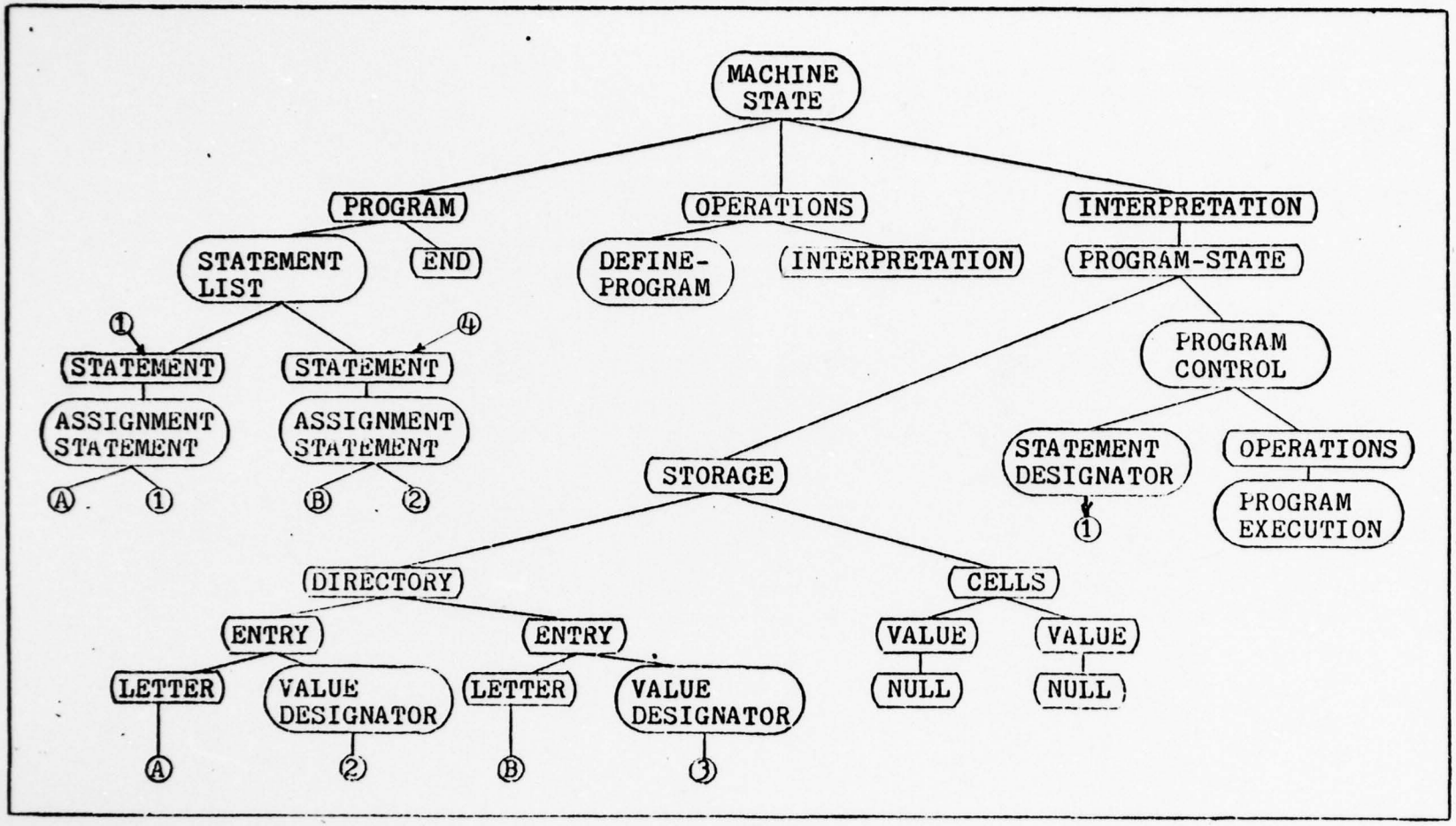


Fig. 37. The machine state during the interpretation phase.

to the original operations tree. When the original operations tree is null the final state, indicating a successful interpretation, has been achieved.

The operation currently in control, PROGRAM-EXECUTION, is listed below:

OPERATION: PROGRAM-EXECUTION

STEP 1. LET st be the <STATEMENT> designated by the <STATEMENT-DESIGNATOR> of the <PROGRAM-STATE>.

STEP 2. PERFORM EXECUTE-ASSIGNMENT-STATEMENT(st)

STEP 3. GO TO STEP 1.

The above operation(PERFORM-EXECUTION) executes steps 1 and 2 until the entire source program has been "interpreted", then control passes to the operations tree on the machine state where the final state is achieved.

After execution of step 2 of PROGRAM-EXECUTION the machine state is transformed. The new PROGRAM-CONTROL tree of the machine state is shown in Fig. 38. Note that the EXECUTE-ASSIGNMENT-STATEMENT operation is now in control. The

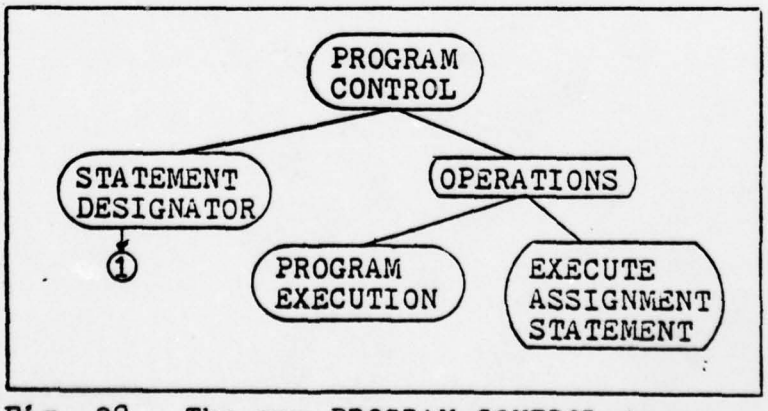


Fig. 38. The new PROGRAM-CONTROL tree

EXECUTE-ASSIGNMENT-STATEMENT operation, listed below, identifies each side of the abstract assignment statement, and then calls upon the ASSIGN operation to make the actual assignment.

```
OPERATION:  EXECUTE-ASSIGNMENT-STATEMENT(st)
            where: st is an ASSIGNMENT-STATEMENT
STEP 1.  LET D be the son of <DIGIT>.
STEP 2.  LET L be the son of <LETTER>
         PERFORM ASSIGN(L,D).
STEP 3.  PERFORM NORMAL-SEQUENCE.
```

After step 2 of the EXECUTE-ASSIGNMENT-STATEMENT operation is executed, the PROGRAM-CONTROL tree of the machine state is changed(shown in Fig. 39). The ASSIGN operation, listed below, is now in control. The first thing the ASSIGN operation must do is determine which storage cell has been saved for the specific letter obtained from the left side of the current assignment statement. This location is determined by the EVALUATE-VALUE-REFERENCE(L) operation. The first

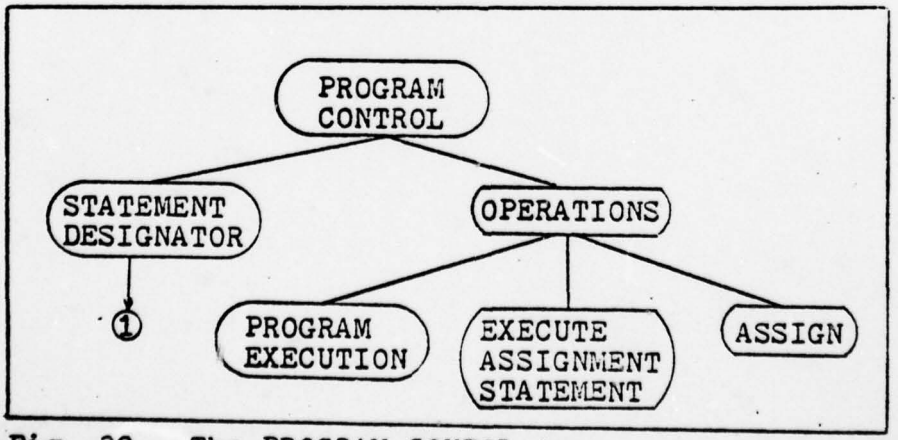


Fig. 39. The PROGRAM-CONTOL tree

statement in the test program has a left side consisting of the letter "A", Fig. 37 reveals that the designator for this letter is "2". Therefore, the EVALUATE-VALUE-REFERENCE operation, when in control, will return the value designator "2" to the ASSIGN operation. The ASSIGN operation will then store the value of the right side of the current assignment statement into the storage cell reserved for the letter "A".

The structure of the PROGRAM-CONTROL tree when the EVALUATE-VALUE-REFERENCE operation is in control is illustrated in Fig. 40. The structure of the PROGRAM-STATE after the ASSIGN operation is complete is illustrated in Fig. 41, note the value of "1" has been stored in node "2" (the node reserved for the value of the letter "A").

With the ASSIGN operation completed and deleted from the PROGRAM-CONTROL tree, control is returned to the EXECUTE-ASSIGNMENT-STATEMENT(st) which immediately transfers control to the NORMAL-SEQUENCE operation, resulting in the PROGRAM-CONTROL tree shown in Fig. 42. The NORMAL-SEQUENCE operation,

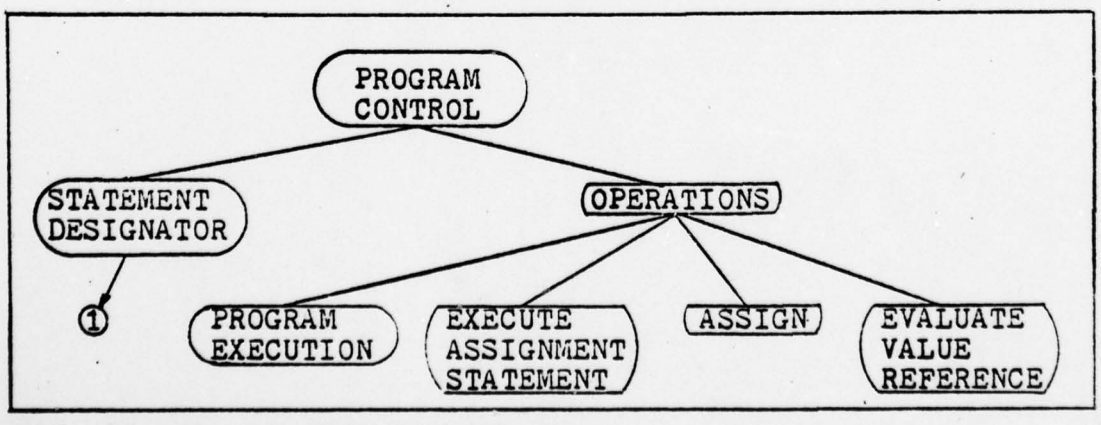


Fig. 40. The PROGRAM-CONTROL tree

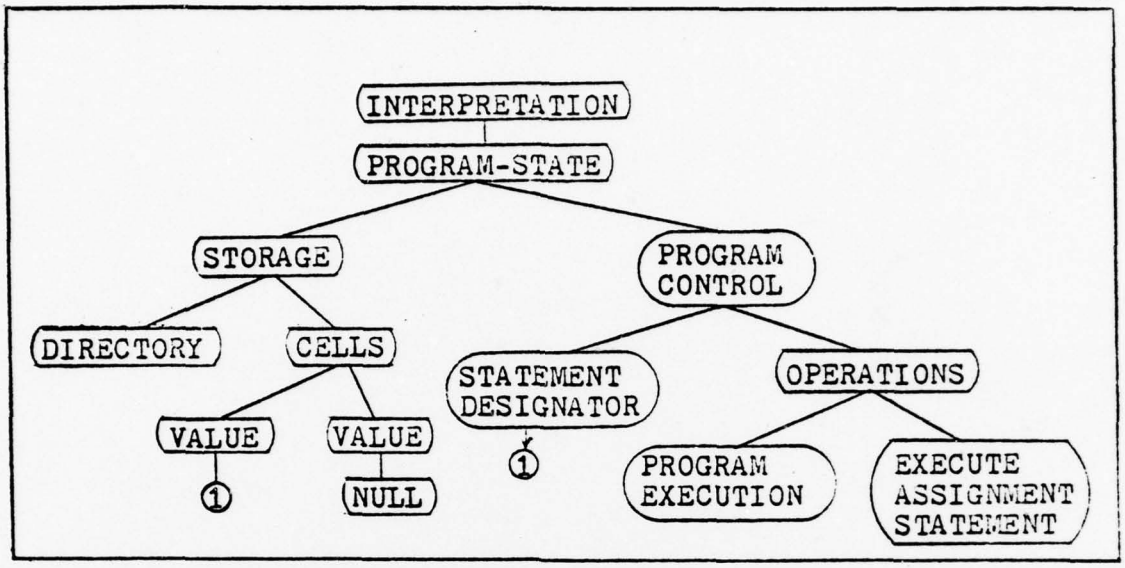


Fig. 41. The structure of the PROGRAM-STATE

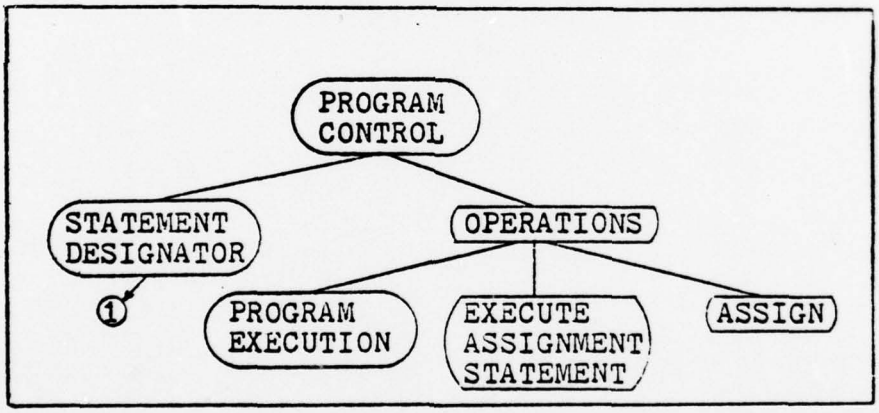


Fig. 42. The PROGRAM-CONTROL tree

given below, advances the STATEMENT-DESIGNATOR, in the PROGRAM-CONTROL tree, so that the designator node points to the next statement to be interpreted. Examining Fig. 37 reveals that the next statement to be executed has a unique name of "4". Therefore, the <STATEMENT-DESIGNATOR> is updated so that it points to node 4. The result of the NORMAL-SEQUENCE

operation is the PROGRAM-CONTROL tree shown in Fig. 43. Since the second statement is also an assignment statement the entire process begins again. However once the second statement is

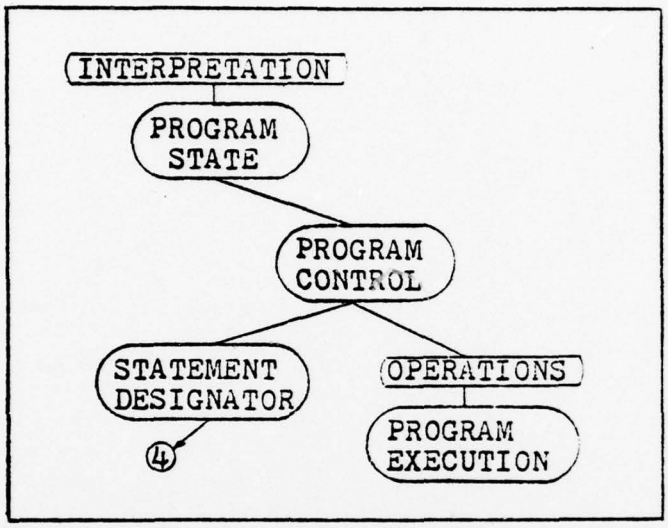


Fig. 43. The PROGRAM-CONTROL tree

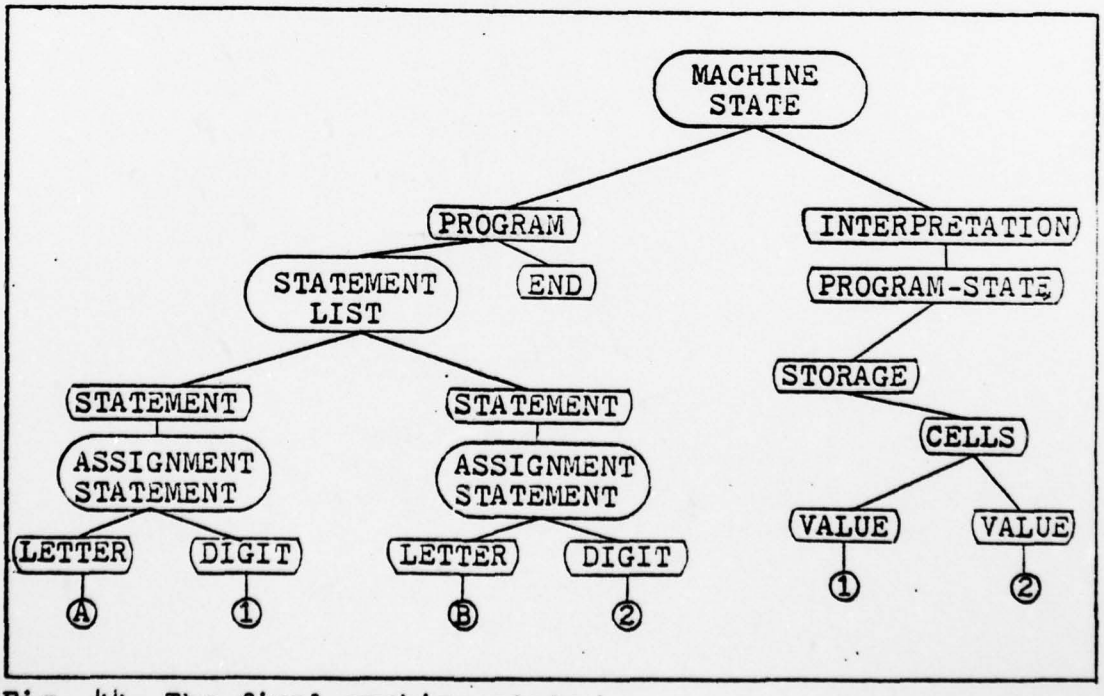


Fig. 44. The final machine state

interpreted the interpretation is completed, the operations tree is null, and the final machine state(Fig. 44) is achieved.

The operations involved in PROGRAM-EXECUTION and discussed above are now listed sequentially so that the reader can follow the process more easily.

OPERATION: PERFORM-EXECUTION

STEP 1. LET st be the <STATEMENT> designated by the <STATEMENT-DESIGNATOR> of the <PROGRAM-STATE>.

STEP 2. PERFORM EXECUTE-ASSIGNMENT-STATEMENT(st).

STEP 3. GO TO 1

OPERATION: EXECUTE-ASSIGNMENT-STATEMENT(st)

STEP 1. LET D be the son of <DIGIT>.

STEP 2. LET L be the son of <LETTER>
PERFORM ASSIGN(L,D).

STEP 3. PERFORM NORMAL-SEQUENCE.

OPERATION: ASSIGN(L,D)
where L is a <LETTER>
D is a <VALUE>

STEP 1. PERFORM EVALUATE-VALUE-REFERENCE(L) to get a <VALUE-DESIGNATOR>, vd.

STEP 2. REPLACE the <VALUE> designated by vd with a copy of D.

OPERATION: EVALUATE-VALUE-REFERENCE(L)
where: L is a <LETTER>
result: a <VALUE-DESIGNATOR>

STEP 1. LET vd be a copy of the <VALUE-DESIGNATOR> component of the <DIRECTORY> that contains a <LETTER> equal to L.

STEP 2. RETURN vd.

OPERATION: NORMAL-SEQUENCE

STEP 1. LET stl be the <STATEMENT-LIST>
LET st be the <STATEMENT> of stl that is designated by the <STATEMTNE-DESIGNATOR>, sd of the <PROGRAM-STATE>.

STEP 2. LET sd designate the <STATEMENT> that immediately follows st in stl.

This chapter has presented a brief informal introduction to BASIS/1-12, a semi-formal definition technique.

The following chapter presents the cataloguing criteria developed in chapter II along with the results of applying this criteria to SEMANOL, the Vienna Definition Language, and BASIS/1-12.

VII Cataloguing

The goal of this chapter is to take the cataloguing criteria, established in chapter II, and apply each question to the formal and semi-formal language definition techniques (SEMANOL, VDL, and BASIS/1-12) discussed in chapters IV, V, and VI. The method used in cataloguing the definition techniques consists of listing the questions of the cataloguing criteria followed by the "answers" for each technique. This method allows each reader to quickly evaluate each formal language definition technique. The cataloguing criteria and the "answers" are listed below:

I CONTROL

A. How does this method process the flow of control for sequences, jumps, loops, and procedure calls?

SEMANOL: For sequences, the user moves from the current-statement position along sequence-of-statements-in-program to the next executable statement node determined by the semantics definition. For jumps the next statement executed is determined by the semantics algorithm, which finds the statement with the proper statement label. With loops an active-block-list is employed, containing a sequence of triples(control variable, value of limit, and value of increment). A global variable is used to determine if the loop is being executed for the first time for initiation(placing

trible on the active-block-list). Procedure calls are handled similar to jumps except before the transfer of control the first executable statement after the "call" statement is placed in a return-point-list (a LIFO stack) to be used in determining the successor of the "return" statement.

VDL: The VDL technique allows two basic methods for specifying control for sequences and loops. For sequences the next statement to be executed can be either specified in the semantics definition, or the next statement in a LIFO stack. If the stack method is used the statement executed is deleted from the stack. The next statement to be executed after a jump instruction is determined by the semantics definition of the jump instruction. Loops can be handled in one of two methods, either breakdown the loop "begin statement" and the loop "end statement" into simple test and jump instructions in the TRANSLATOR phase, or use a LIFO stack to determine where to return after executing the loop "end statement" (NEXT, CONTINUE, etc.). Procedure calls in the VDL are handled with a LIFO stack containing the location of the next executable statement following the "call" statement, thus saving the return point for the "return" instruction.

BASIS/1-12: For sequences a pointer(the statement designator node) is incremented to the next statement on the program tree. The TRANSLATOR transforms the

"jump" instructions so that they contain the unique names of the nodes to which the jump is to be made. Therefore, when the INTERPRETER executes the jump command the statement designator node is made to point to the node specified in the jump command. BASIS/1-12 can process loops in one of two methods. The TRANSLATOR can break down the loop control statements into simple test and jump statements, or the INTERPRETER can process the loop control statements using the semantic definitions specifying the value of the statement designator node. Procedure calls are processed with the INTERPRETER which manipulates the statement designator node into pointing to the procedure called.

II MEMORY

B. How is memory defined? (This question refers to how variables are defined.)

SEMANOL: In SEMANOL the memory is defined as a single level associative memory, represented as name-value pairs.

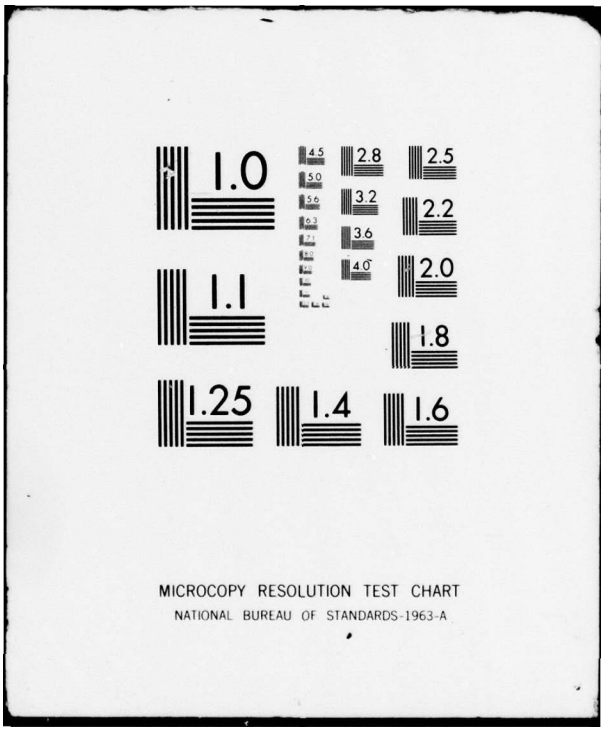
VDL: In the VDL memory is defined with three symbol tables; the environment, denotation, and attribute tables. The environment table specifies the machine location for the source program variable. The denotation table maintains the proper value for each machine location. The attribute table maintains the machine location and its attributes(type: real, integer, etc.). In its simplest representation the memory is a selector-object pair with

AD-A052 917 AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OHIO SCH--ETC F/G 9/2
A CATEGORIZATION AND EVALUATION OF FORMAL AND SEMI-FORMAL DEFIN--ETC(U)
MAR 78 B D GUILMAIN

UNCLASSIFIED AFIT/GE/MA/78M-1 NL

2 OF 2
AD A052917

END
DATE FILMED
5-78
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

the variable as the selector and the value as the object.

BASIS/1-12: In BASIS/1-12 the memory is defined as consisting of a "directory" and storage locations. The directory consists of a variable and a value designator node. The value designator node points to the location(unique name) allocated for that variable.

III EXPRESSIONS

C. How are evaluations of expressions defined?

SEMANOL: The constants and operators of the defined language are translated into SEMANOL constants and operators and then the INTERPRETER, operated by algorithms written in SEMANOL, performs the required operations on the SEMANOL representations of the constants and operators.

VDL: The expressions are represented as abstract objects on a control tree. The expressions are evaluated in accordance with commands in a algorithm that expands or contracts the control tree.

BASIS/1-12: Expressions are represented as abstract objects on a program tree. Rules for evaluating the expressions are given in algorithms written in English.

D. How are lexical transformations processed?

A lexical analysis of the source program is basically performed the same way for SEMANOL, the VDL, and BASIS/1-12. The lexical analysis is the first processed performed on the source program. The analysis is performed by an algorithm which performs lexical transformations(macro

substitutions), removes comments from the source program, and removes text as required by "skip" instructions.

E. Can external functions and relations be called from a system library?

All three techniques SEMANOL, the VDL, and BASIS/1-12 can specify functions and relations of the system library, however the VDL and BASIS/1-12 are not as yet implemented on host machines.

F. How is the semantic operator expressed?

SEMANOL: In SEMANOL the semantic operator is expressed as a SEMANOL program(algorithm).

VDL: The semantic operator in the VDL is expressed as value returning instructions which manipulate a control tree.

BASIS/1-12: In BASIS/1-12 the semantic operator is expressed as an algorithm written in English that manipulates an operations tree.

IV CLARITY

G. Can people understand the method?

All three techniques can be readily understood by people. However, the author found the SEMANOL definition easiest to follow, the BASIS/1-12 technique was the next easiest to follow, leaving the VDL technique as the most difficult technique to follow.

H. Is high level expressiveness utilized? (How much detail must one know before utilizing the technique?)

SEMANOL: High level expressiveness is utilized in SEMANOL. The SEMANOL algorithms are straightforward and easy to follow.

VDL: No, the user must keep careful track of each minute detail to follow the definition.

BASIS/1-12: Yes, the BASIS/1-12 instructions, written in English, can be followed without a "strict" maintenance of the machine state.

I. Are the mneumonic names helpful to the reader?

The mneumonic names used in each of the three techniques are helpful to the reader. Conventional notation found in mathematics and programming languages is used, enabling a user to easily interpret the meaning of the term used.

J. Does this method seperate context-free syntax, context-sensitive syntax, and the semantic parts of a language definition?

SEMANOL: Yes, through differnet steps in the interpretation.

VDL and BASIS/1-12: Yes, through the use of three seperate abstract machines. The ANALYZER checks the context-free syntax, the TRANSLATOR checks the context-sensitive syntax, and the INTERPRETER defines the semantics.

V COMPLETENESS

K. Does this technique provide a complete definition?

Each of the three techniques SEMANOL, the VDL, and BASIS/1-12 provides a complete definition, the techniques can describe every possible situation in the language being defined.

VI CONTEXT-FREE AND CONTEXT-SENSITIVE SYNTAX

L. How is the context-free syntax processed?

SEMANOL: A context-free parse is conducted by an algorithm which parses the source program in accordance with the context-free syntax specified. The result is a context-free parse tree of the source program.

VDL and BASIS/1-12: The last function of the ANALYZER is to perform a context-free parse on the source program using the context-free syntax rules specified. The result, a context-free parse tree, is passed to the TRANSLATOR.

M. How is the context-sensitive syntax processed?

SEMANOL: A SEMANOL algorithm is applied to the context-free parse tree to determine if the context-sensitive restrictions are satisfied.

VDL: The TRANSLATOR performs the context-sensitive checks on the context-free parse tree, supplied by the ANALYZER. The context-sensitive checks are performed by applying a series of conditional statements to the context-free parse tree.

BASIS.1-12: The TRANSLATOR perofrms the context-

sensitive checks on the context-free parse tree, supplied by the ANALYZER. The context-sensitive checks are performed by operations that conduct context-sensitive tests.

VII DEFINING THE EXECUTION OF A PROGRAM

N. How does this method define the execution of a program?

SEMANOL: The definition of the execution of a program is program oriented. The consequences of a computation are described in a series of semantic definition, expressed as SEMANOL programs.

VDL and BASIS/1-12: The execution of a program is defined by a series of state transitions. Each state transition transforms the machine state until the control tree(operations tree for BASIS/1-12) is empty. The empty control(operations) tree characterizes the final state.

VIII DECIDING THE VALIDITY OF A PROGRAM

O. What constitutes a valid program?

SEMANOL: A program is valid if it passes a context-free parse, a context-sensitive check, and a successful semantic execution.

VDL: A program is valid if it passes a context-free parse, a context-sensitive check, and an execution which results in an empty control tree.

BASIS/1-12: A program is valid if it passes a context-free parse, a context-sensitive check, and an execution

which results in an empty operations tree.

IX SPECIFICATION OF IMPLEMENTATION DEPENDENCIES

P. How are the implementation dependencies defined?

SEMANOL: Machine dependencies are stated in a seperate section of the language definition. This section is accessed through parameters used in the semantic definitions.

VDL: Machine dependencies are implemented through variables in the semantic definitions of instructions.

BASIS/1-12: Machine dependencies are implemented through variables in the operations.

Q. Are the representations of the data types and operators machine independent?

SEMANOL: Yes, the technique can be used without forcing the limitations of the host machine on the definition.

VDL and BASIS/1-12: Yes, these techniques are not implemented on an actual machine.

A summary of the information presented above is given in Table I. A comparative cataloguing of the three techniques; SEMANOL, the VDL, and BASIS/1-12 is given in table II on page 94. A numbering system was developed to permit a comparative evaluation of each technique. In this numbering system the number "1" represents the easiest or best situation, while the number "3" represents the hardest or worst situation. The assignment of the same number to two techniques represents

Table I

Summary of answers for the cataloguing criteria

AREA OF CONCERN	SEMANOL	VDL	BASIS/1-12
I CONTROL	Determined by the semantics definition.	Determined by the semantics definition or a LIFO stack.	Determined by a designator node, used as a pointer.
II MEMORY	Single level associative memory.	Defined with three symbol tables.(variable,value) pairs.	Defined with a directory and storage locations.
III EXPRESSIONS	Constants and operators are translated into SEMANOL representations.	Represented as abstract objects.	Represented as abstract objects.
IV CLARITY	The easiest method to understand. The clearest.	The hardest method to follow.	The second clearest
V COMPLETENESS	Capable of describing every situation	Same as SEMANOL.	Same as SEMANOL.
VI CONTEXT-FREE AND CONTEXT-SENSITIVE SYNTAX	Context-free parse performed by an algorithm. Context-sensitive syntax determined by an algorithm testing each restriction.	Context-free parse performed by the ANALYZER. Context-sensitive syntax checks performed by the TRANSLATOR.	Same as the VDL.

Table I continued

Summary of answers for the cataloguing criteria

AREA OF CONCERN	SEMANOL	VDL	BASIS/1-12
VII DEFINING THE EXECUTION OF A PROGRAM	The definition is program oriented.	The definition is characterized by a series of state transitions.	Same as the VDL.
VIII DECIDING THE VALIDITY OF A PROGRAM	A program is valid if it has good context-free and context-sensitive syntax, and a successful semantic execution.	A program is valid if it has good context-free and context-sensitive syntax, and the final state is achieved.	Same as the VDL.
IX SPECIFICATION OF IMPLEMENTATION DEPENDENCIES	Implementation dependencies are stated in a separate section, accessed through parameters used in the semantic definitions.	Dependencies are implemented through variables in the semantic definitions. This method is not implemented.	Dependencies are implemented through variables in the operations. This method is not implemented.

Table II

Comparative evaluation of SEMANOL, VDL, and BASIS/1-12

CHARACTERISTICS	SEMANOL	VDL	BASIS
I Follow the flow of control	1	3	1
II Visualize the memory scheme	1	1	2
III Follow the evaluation of an expression	1	3	2
IV The clarity of the language definition	1	3	2
V Provides a complete definition	1	2	2
VI Follow the contex-free and context-sensitive syntax	1	2	3
VII Follow the execution of a source program	1	3	2
VIII Determine the validity of a source program	1	2	2
IX Recognize implementation dependencies	1	3	2

a situation that presented an equal degree of difficulty to the author.

Table II is divided into nine sections. Each section evaluates SEMANOL, the VDL, and BASIS/1-12 in one of the nine important qualities required of a formal definition technique; control, memory, expressions, clarity, completeness, context-free and context-sensitive syntax, execution of a source program, validity, and implementation dependencies. A brief justification of the ratings in each section is given in the paragraphs below.

The first section, follow the flow of control, rates SEMANOL and BASIS/1-12 equal and superior to the VDL. The SEMANOL technique specifies the flow of control in semantic definitions. In BASIS/1-12 the flow of control is determined through operations which affect the value of the statement designator node. The VDL technique relies on the status of the machine state, transfering control through VDL instructions.

The memory scheme is evaluated in section two. In this section SEMANOL and the VDL are rated equal and superior to BASIS/1-12. In SEMANOL and the VDL memory is basically viewed as (variable, value) pairs. In BASIS/1-12 memory is defined with pointers and storage locations.

Section III deals with the evaluation of expressions. It is easier to follow the evaluation of an expression in SEMANOL because there is no tree manipulation. Therefore SEMANOL is rated higher than the VDL or BASIS/1-12. BASIS/1-12 is rated higher than the VDL because BASIS/1-12 utilizes

algorithms, written in English, to define the state transitions that compose the evaluation of an expression. In the VDL, expressions are evaluated through instructions that depend heavily on the machine state, a tree structure.

The major factor in determining the clarity of a language definition, section IV, is the dependence on a parse tree. Since SEMANOL minimizes this dependence it is rated as being clearer than BASIS/1-12 and the VDL. BASIS/1-12 is rated higher than the VDL because the operations, written in English, in BASIS/1-12 help minimize the dependence on the parse tree. The VDL instructions require the machine state, and thus this technique relies the most on the parse tree.

The fifth section evaluates the ability of a technique to provide a complete definition. All the techniques examined were complete in that they could define any situation which could occur within the defined language. However SEMANOL is successfully implemented on an actual machine, thus proving that it is complete.

The technique used to specify the context-free and the context-sensitive syntax is evaluated in section VI. SEMANOL is rated as the best method because of the meta-language, similar to BNF, used in the SEMANOL technique. The conditional statements used in the VDL to check the syntax makes the VDL superior to BASIS/1-12. BASIS/1-12 is rated the lowest because the context-free syntax is divided into high and low level syntax.

Section VII deals with the execution of a source program. In this section SEMANOL is rated as the best because no tree manipulation is required. BASIS/1-12 is rated second because its operations, written in English, allow for easier tree manipulation than is possible in the VDL. The VDL executes a program through instructions that rely heavily on the machine state, a tree structure.

In section VIII SEMANOL was rated superior to the VDL and BASIS/1-12. In determining the validity of a source program in the SEMANOL definition the user does not have to maintain a changing machine state in search of the final state. The validity of a source program is dependent on attaining the final state when the VDL or BASIS/1-12 technique is utilized, therefore the VDL and BASIS/1-12 are rated as being equal.

The final section, section IX, pertains to implementattion dependencies. Again the SEMANOL technique ranks first because the machine dependencies are specified in a seperate section accessed through paramenters, and the SEMANOL technique has been proven through actual implementation. The BASIS/1-12 technique is rated higher than the VDL because machine dependencies are implemented through variables in the operations, while the VDL technique implements machine dependencies through variables in the required instructions.

This chapter applied the questions, comprising the cataloguing criteria established in chapter II, to the three techniques; SEMANOL, the VDL, and BASIS/1-12 covered in chapters IV, V, and VI, respectively. The author's comparative

evaluation of each technique was also presented.

Chapter VIII, the conclusion, presents advantages and disadvantages of each technique along with the author's choice of the "best" definition technique.

VIII Conclusion

This thesis examined three techniques for specifying formal definitions of programming languages; SEMANOL, the Vienna Definition Language(VDL), and BASIS/1-12. Some of the advantages and disadvantages of each technique are specified below.

SEMANOL's close resemblance to programming languages makes it fairly easy to learn except for the fact that information on SEMANOL is spread out over several publications, and no "simple" example is "talked through" in any of the publications. While a parse tree is utilized in the SEMANOL technique, execution of a source program statement can be easily followed without a parse tree. The SEMANOL technique is also easy to learn because the user can readily visualize his progress when executing a source program. Once learned, the SEMANOL algorithms are easy to follow, and the user does not have to keep track of a changing tree structure.

Information on the VDL does not seem to be as spread out as the information on SEMANOL. Another advantage of the VDL is that there are only two catagories of tree manipulating instructions. The major disadvantage of the VDL is that the user must keep track of every minute change in the control tree, and the execution of all but the simplest programs requires an unreasonable number of machine states. Another characteristic of the VDL which the author considered a disadvantage, at first, was the flexibility of the VDL. Different authors

describing the same situation using the VDL used completely different approaches which seemed to contradict each other, until a detailed examination was performed. The overall description of the abstract machines used by the VDL definition also varied from author to author. One author described the VDL technique as consisting of two abstract machines, a TRANSLATOR, and an INTERPRETER. Other authors described the VDL technique as consisting of three abstract machines, an ANALYZER, a TRANSLATOR, and an INTERPRETER. The difference between these two viewpoints is that the TRANSLATOR in the two machine description performs the same functions as the ANALYZER and the TRANSLATOR in the three machine description.

After studying SEMANOL and the VDL, the author found BASIS/1-12 relatively easy to comprehend. Some of the advantages of BASIS/1-12 are that the algorithms for the state transitions are written in English and are very easy to follow. Because the algorithms are easy to follow, the problems of keeping track of the machine states are not as severe as in the VDL. However, the user must keep track of certain instructions(ATTACH and DELETE) as their effect on the machine state varies with the syntax of the source program.

The technique that provides the clearest definition of a source program is SEMANOL, followed by BASIS/1-12 and the VDL, in that order. The major reasons for SEMANOL s clearity is that the user does not have to trace state transitions and machine states. BASIS/1-12 is rated superior to the VDL because the BASIS/1-12 algorithms, written in English, are

simpler than the VDL instructions when trying to manipulate a tree structure.

From the comparisons in Table II, page 94, and the above described advantages and disadvantages an overall rating of the three techniques would have to list SEMANOL as the best technique followed by BASIS/1-12 and the VDL, in that order.

The comparisons of the techniques, SEMANOL, the VDL, and BASIS/1-12, and the list of advantages and disadvantages of each technique makes this thesis the only single source for a reader interested in understanding the mechanics of each technique.

Recommendations

While this thesis covered only interpreter oriented formal and semi-formal definition techniques a similar report covering mathematical techniques is needed. The purpose of the report besides cataloguing certain techniques should be to provide simple examples, demonstrating how the techniques work; so that "new" explorers into this area can easily see how a method works and then study the details of each technique.

As any field of study develops the rules governing that field become more precise, rigid, and definite. As man's communication with his fellow man developed, the need for precise, rigid and definite rules brought about grammer rules and the dictionary. The same is true with computer programming languages. Computer programming languages have developed to the point where standard definitions are needed. The first

step toward standard definitions is to develop a formal definition technique(a meta-language) that provides useful information to all interested individuals.

"The meta-language of a formal definition must not become a language known to only the high priest of the cult. Tempering science with magic is a sure way to return to the dark ages"(Ref 3).

Bibliography

1. Robinson, J.A. The Definition of Programming Languages technical report. Syracuse, Syracuse University, Oct 1976.

2. Berning, Paul T. An Introduction to SEMANOL. Redondo Beach: TRW Defense and Space Systems Group, April 1977.

3. Marcotty, Michael et al. "A Sampler of Formal Definitions," Computing Surveys, 8: 191-276(June 1976).

4. Anderson, E.R. et al. Issues in the Formal Specification of Programming Languages: TRW Defense and Space Systems Group, June 1977.

5. Wegner, Peter. "The Vienna Definition Language," Computing Surveys, 4: 5-63(March 1972).

6. Carlson, Carl R. A Study of the Applicability of the Vienna Definition Language to the Specification of the Semantics of the Next Event Simulation Concept. technical report No 59 for the office of Naval Research. Iowa City, University of Iowa, August 1972. (AD-746-833).

7. Anderson, E.R. et al. A Metalanguage for Programming the Semantics of Programming Languages. Redondo Beach:TRW Defense and Space Systems Group, June 1974.

8. Berning, Paul T. A SEMANOL(76) Specification of Minimal Basic. Final technical report No. RADC-TR-77-170. Rendondo Beach, TRW Defense and Space Systems Group, May 1977.

9. Berning, Paul T. SEMANOL(73) Reference Manual. Final technical report No. RADC-TR-75-211. Rendondo Beach, TRW Defense and Space Systems Group, June 1975.

10. Hoare, C.A.R. "An Axiomatic Basis for Computer Programming," Communication of the ACM, 12: 576-583(October 1969).

11. Landin, P.J. "The Mechanical Evaluation of Expressions," Computer Journal, 6: 308-320(1964).

12. Lee, J.A.N. Computer Semantics: Studies of Algorithms Processors, and Languages. New York, Cincinnati, Toronto, London, Melbourne: Van Nostrand Reinhold Co, 1972.

13. Anderson, E.R. et al. SEMANOL(76) Interpreter Documentation. Final technical report No. RADC-TR-77-365. Redondo Beach, TRW Defense and Space Systems Group, November 1977.

14. Berning, Paul T. A SEMANOL(73) Implementation Standard for Jovial(J73). Final technical report No. RADC-TR- 75-211. Redondo Beach, TRW Defense and Space Systems Group, June 1975.

15. Gries, D. Compiler Construction for Digital Computers. New York, London, Sydney, Toronto: John Wiley and Sons Inc, 1971.

16. Marcotty, Michael et al. "The Definition Mechanism for Standard PL/I," IEEE Transaction on Software Engineering, 3: 416-450(November 1977).

Vita

Bruce Daniel Guilmain was born in Manchester, New Hampshire, on June 16, 1950. At the age of two his family moved to Nashua, New Hampshire. Upon graduation from Bishop Guertin High School, Nashua, New Hampshire, in 1968, Bruce Guilmain attended Lowell Technological Institute, Lowell, Massachusetts. In June 1972, Bruce Guilmain graduated with a Bachelor of Science degree in electrical engineering and was commissioned a Second Lieutenant in the United States Air Force. Second Lieutenant Guilmain served his initial tour of duty as a deputy missile combat crew commander, at Malmstrom AFB, Montana. First Lieutenant Guilmain was married to Billie Carrol Anderson on May 2, 1975; they now have one son, Jason. In September 1975, while at Malmstrom AFB, First Lieutenant Guilmain was upgraded to missile combat crew commander. Captain Guilmain entered the Air Force Institute of Technology at Wright-Patterson AFB, Ohio, in September 1976 to begin study toward a masters degree in electrical engineering.

Permanent address: 323 Main Dunstable Road
Nashua, New Hampshire 03060

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFIT/GE/MA/78M-1	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A CATEGORIZATION AND EVALUATION OF FORMAL AND SEMI-FORMAL DEFINITION TECHNIQUES		5. TYPE OF REPORT & PERIOD COVERED MS Thesis 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Bruce D. Guilmain		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Air Force Institute of Technology(AFIT-EN) Wright-Patterson AFB, Ohio 45433		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center(RADC) Information Sciences Division(ISIS) Griffiss AFB N.Y. 13441		12. REPORT DATE March 1978 13. NUMBER OF PAGES 116
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

Approved for public release; IAW AFR 190-17

JERRAL F. GUESS, Captain, USAF
Director of Information

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Meta-language
semantics
SEMANOL
Vienna Definition Language
BASIS/1-12
categorization

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Operational techniques for defining computer programming languages are examined; specifically, SEMANOL, the Vienna Definition Language(VDL), and BASIS/1-12. A survey of the operational methods is given, in which specific examples of SEMANOL, the VDL, and BASIS/1-12 are explained in detail. A cataloguing criteria is established and evaluated. The cataloguing criteria is then used to categorize and evaluate SEMANOL, the VDL, and BASIS/1-12. The SEMANOL technique was judged as the best technique followed by BASIS/1-12 and the VDL, in that order.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE